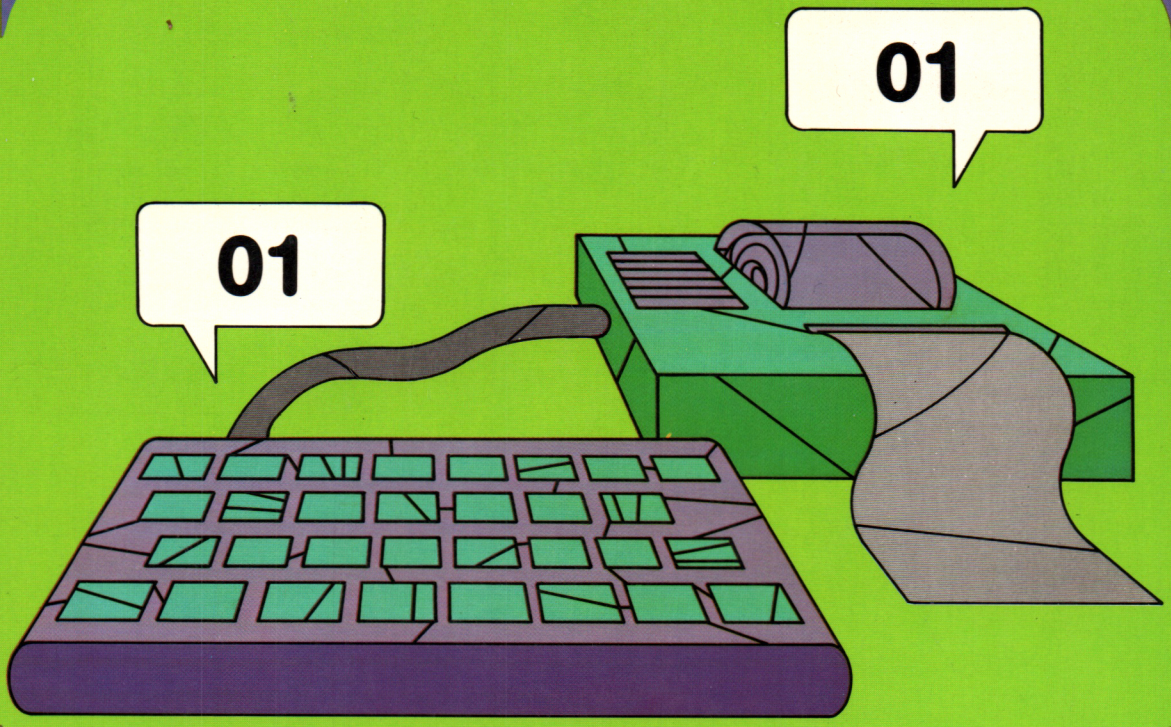
T/S 2068 T/S 1500 T/S 1000

Timex Sinclair 2068, 1500 and 1000 Machine Language Programming and Interfacing

Joseph J. Carr

Your key to more powerful, faster running programs that use *less* memory!



BookWare

For the Timex Sinclair 2068, 1500, 1000 and ZX-81 Computers

TIMEX/SINCLAIR 2068, 1500 AND 1000
MACHINE LANGUAGE PROGRAMMING AND INTERFACING

#9.95

TIMEX/SINCLAIR 2068, 1500 AND 1000 MACHINE LANGUAGE PROGRAMMING AND INTERFACING

JOSEPH J. CARR

A Reston Computer Group Book
Reston Publishing Company, Inc., Reston, Virginia
A Prentice-Hall Company

Library of Congress Cataloging in Publication Data

Carr, Joseph J.
Timex/Sinclair 2068, 1500, and 1000 machine language programming and interfacing.

Includes index.
1. Timex Sinclair 2068 (Computer)—Programming. 2. Timex Sinclair 1500 (Computer)—Programming. 3. Timex 1000 (Computer)—Programming. 4. Programming languages (Electronic computers) I. Title.
QA76.8.T494C37 1984 001.64'2 84-9774
ISBN 0-8359-7699-8

© 1984 by
Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia 22090

All rights reserved. No part of this book
may be reproduced in any way, or by any means,
without permission in writing from the publisher.

10 9 8 7 6 5 4 3 2 1

PRINTED IN THE UNITED STATES OF AMERICA

CONTENTS

Preface *vii*

PART I GETTING STARTED 1

Chapter 1 Your Timex Computer 3
Chapter 2 Introduction to Processors and Processing 12
Chapter 3 Setting Up the Timex Computers 24
Chapter 4 Using the Timex Computers 33

PART II INTERFACING HARDWARE 37

Chapter 5 Z80 Architecture 39
Chapter 6 Z80 I/O 61
Chapter 7 Interfacing Memory to the Z80 87
Chapter 8 Interrupt Hardware 100
Chapter 9 Interfacing Peripherals 110
Chapter 10 External Power Supplies and "Glitch Witches" 121

Chapter 11 Timex Interface Connector 127
Chapter 12 Data Conversion: A/D—D/A 139
Chapter 13 Controlling External Devices and Circuits 171

PART III MACHINE LANGUAGE PROGRAMMING 181

Chapter 14 Z80 Addressing Modes 183
Chapter 15 The Z80 Instruction Set (General) 191
Chapter 16 Servicing Interrupts 199
Chapter 17 Arithmetic/Logic Operations 211
Chapter 18 Machine Language Programming of the T/S-1000 220
Chapter 19 Details of the Z80 Instruction Set 223

Appendix A Miscellaneous Z80 Information 239
Appendix B Z80 Family Support Chips 255

Index 267

PREFACE

As a writer on books dealing with popular computing and electronics topics, I noted with more than casual interest that the *Timex* T/S-1000 microcomputer was being sold at discount stores such as *K-Mart* and in Washington's *Circuit City* stores. A colleague of mine owned one of the machines and extolled its virtues seemingly endlessly . . . so the little T/S-1000 was on my mind quite a bit. After I gave my colleague a complementary copy of my book *Z80 User's Manual* (Reston, Va: Reston Publishing Co., 1980) he remarked that it contained a lot of information and would make a good T/S-1000 book. That statement talked me into buying a T/S-1000, so I went to *K-Mart* with checkbook in hand to get a first hands-on look at the T/S-1000.

It is my habit to read the instruction manual of any new appliance completely before attempting to work the thing. I once made my reputation as an engineer in a hospital operating room during surgery when a critical piece of equipment was "malfunctioning." I simply read the instruction manual and set the switches according to the book . . . no one else had read the manual and had assumed that the new one operated like the older model. Ever since that time, reading of instruction manuals has been my habit. The T/S-1000 manual contained a very happy surprise for me. On page 123, at the beginning of Chapter 25 ("Using Machine Code"), they

recommend my *Z80 User's Manual!* This is the very book which I had given to my friend.

In the next few weeks, I played with the T/S-1000 extensively, and found that it was not a toy, but a serious and very powerful little computer. That explains the fact that there has grown up around the T/S-1000, and its Sinclair marketed predecessor (same machine in different clothes) the ZX-81, a large cottage industry of firms and people offering software, hardware and other assorted thingameetights for these computers. The T/S-1000 not only serves as an excellent introduction to computing, but will serve the user for some time to come after he or she is more expert.

The Timex T/S-1000 was succeeded by the T/S-1500 and the T/S-2068 personal computers. The T/S-1500 was a slightly more advanced T/S-1000 that used a rubbery keyboard that was quite an improvement over the original keyboard. The T/S-2068 was a complete departure from the original design. The T/S-2068 has an expanded interface connector, a Command Cartridges connector and both RF and video monitor outputs. In addition, it has fully color capability. All in all, the T/S-2068 computer is a very advanced machine at low cost.

JOSEPH J. CARR, MSEE

TIMEX/SINCLAIR 2068, 1500 AND 1000
MACHINE LANGUAGE PROGRAMMING AND INTERFACING

PART I
GETTING STARTED

CHAPTER I

YOUR TIMEX COMPUTER

Having owned a "real" microcomputer or two, I was tempted at first to ignore the little Sinclair ZX-80 kit-form computer being sold by mail. But as more and more people bought them, the word filtered into my thick head that this was a *serious* microcomputer. Then they came out with the ZX-81, and the identical Timex version, the T/S-1000, and my change of heart was confirmed. The ZX-81 and T/S-1000 are serious computers. Since they are the same, incidentally, I will refer to them by the T/S-1000 designation from here on out.

The T/S-1000 computer (see Fig. 1-1) is one of the best trainer units that I have seen. For a very low cost, the student just breaking into computers can obtain a machine which can be programmed in BASIC and machine language (one vendor also offers an Assembly Language tape for it), yet is very simple to operate. The experienced user may find some features that he or she doesn't like, but that is the price one pays for low cost. The strange keyboard is probably the best example. The keyboard is small, and is of the membrane type so there is no tactile feedback to let you know that the keystroke "took."

Since the introduction of the Timex T/S-1000 computer in the United States, there have been two new Timex/Sinclair computers. The model T/S-1500 is an updated version of the T/S-1000, while the T/S-2068 is a much more sophisticated "color computer" that retains the features of the earlier models, while adding considerable new capability.

One group of users which may easily overlook the little TIMEX's is

(a)

(b)

FIGURE 1-1 (a) Timex T/S-1000, oblique view. (b) Timex T/S-1000, top view.

scientific and other students in university laboratories. While many well-funded research projects are equipped with rooms full of DEC PDP-11 computers and accessories, many, perhaps most, researchers have to make do. Because of the interface connector on the back of the TIMEX computers, and the types of accessories that can be bought from various vendors, this machine is ideal for the researcher who needs only basic computing and/or data acquisition, and doesn't have a large grant to fill the bill. Any of the TIMEX computers can be used to control the experiment, acquire data and then massage the data as a routine number cruncher. The data storage is crude, i.e., an audio cassette tape recorder, but is sufficient for this use. There is no reason why someone cannot come up with a disc driver for this computer, and I expect that we will see it on the market sometime in the future.

TIMEX T/S-1000

The T/S-1000 computer uses an ordinary television set as the display. There is a built-in RF modulator that will output the video signal on either Channel-2 or Channel-3, depending upon the setting of a slide switch on the bottom of the T/S-1000 cabinet. The need for two channels is evident when one tries to play the computer into a TV set that has a strong local station on the same channel. The isolation is poor, so it is necessary to use an unoccupied channel for the computer. The U.S. Federal Communications Commission is not in the habit of assigning adjacent channels in the low VHF region to broadcasters within the same community, so it is unlikely that you will find both channel-2 and channel-3 are used in your area.

The T/S-1000 has a built-in video display generator that feeds the TV modulator. The display is 32 characters wide by 24 lines; only 22 lines are usable, however, because the operating system requires the bottom two lines for its own purposes. Thus, the user can print up to 704 characters or spaces on the screen at any one time.

The cost of the T/S-1000 is $99.95 at retail, but discounts are common. I bought mine from a K-Mart store for $77, and the 16K memory module (see Fig. 1-2) for $49.95. An updated version, called the TIMEX 2068 is now on the market.

Mass storage is by audio cassette tape player. The binary digits of the computer are converted to audio tones, and then output to the cassette recorder. On read operations, the audio tones of the tape are reconverted into 1's and 0's of the binary system so that the computer can understand what is being said. Although sales people will tell you that any cassette player will work, I find that is a little optimistic. The cassette player should be one of

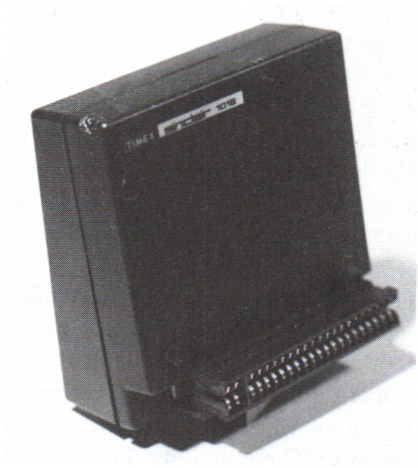


FIGURE 1-2. 16K add-on memory module.

reasonable quality so that the speed is accurately maintained. Also, it should have a counter. That feature will make it easier to locate programs on a tape which contains many programs. Otherwise, you might be faced with a frustrating chore of searching an entire C60 in order to find a specific program. If you have a counter, and are smart enough to keep records of the index number where each program is located, then it becomes a lot easier. Also, be certain that the cassette player has 3.5 mm jacks (normal size) for external earphone and microphone. The clerk who sold me my T/S-1000 offered a model which used an internal microphone, with no provision for external microphone. *That* makes it useless to record programs from the T/S-1000! The clerk told me that she had been selling this model with the Timex computer "for several weeks, since Christmas when we ran out of another model."

T/S-1000 FEATURES

The T/S-1000 is based on the popular Z80 microprocessor chip, and operates at a clock frequency of 3.5 mHz. On board is an 8K read only memory (ROM) that contains the BASIC interpreter and the operating system, and a 2K random access (read/write) memory (RAM). Optionally, one can buy from Timex a 16K memory module which brings total memory up to 18K. Other vendors offer memory modules to 64K. Keep in mind, however, that not all of the T/S-1000 memory is available to the user. The display program requires a total of 727 bytes, while the BASIC interpreter wants 124 bytes for its own use, for a total of 851 bytes. This means that only 1197 bytes are available to the user in the basic T/S-1000, or 17,581 if the 16K memory module is used. Since the earlier ZX-81 device contains only 1K of internal memory, that computer is affected even more by this limitation.

Also be aware that the display section of RAM is not hard allocated. It occupies as much space as needed by the actual characters to be displayed. While this can be a problem in some cases, it is also possible to use this fact to advantage when there is little data to display, but we need extra memory. Provided that the programmer takes pains to ensure that the display does not ever overwrite the program, then the program can use this otherwise allocated memory space.

The keyboard on the Timex T/S-1000 is a membrane-type with 40 keys. Each key is used to perform up to four functions: alphanumeric character, a function, a graphics symbol and a BASIC keyword.

Programming in BASIC is a little different on the T/S-1000, as compared with other computers. In order to achieve the small size, the computer uses keywords for the BASIC commands. For example, when you want to write a PRINT command on most microcomputers, you would type out the letters P-R-I-N-T, but on the T/S-1000 you would press the "PRINT"

keyword, which shares the same key with the 'P' character, (") character—i.e. shifted-P—and the TAB function (another BASIC command). This multiple use of keys makes the overall keyboard a lot smaller. More will be said of the functioning of the keyboard in Chapter 4 when we discuss Using the T/S-1000.

T/S-1500

The Timex/Sinclair T/S-1500 (Fig. 1-3) is basically an updated T/S-1000, with certain key differences. Because of the similarities, however, you are encouraged to also read the preceding section on the T/S-1000. All three Timex computers use the Zilog-developed Z80A microprocessor chip as the central processing unit. The most obvious differences between the 1500 and the 1000 are the shape, color and keyboard of the computer. The new color (which is also used on the T/S-12068) is a silver-gray and is somewhat more attractive than the old black color of the 1000. The newer 1500 is wider and shallower than the 1000.

The new 1500 keyboard uses "chicklet" pushbutton keys rather than the old membrane style of keyboard used on the 1000. The keyboard was one of the few common complaints heard on the 1000. Many people did not like the non-tactile membrane keys because there is no "feel" feedback when a button is pressed. It was possible to "double-strike" a key for the lack of tactile feedback. The newer 1500 computer, on the other hand, uses "real" pushbuttons that actually move when pressed. The feel is a soft, rubbery effect that is easy to use. In general, there are more functions on the 1500 keys than on the 1000 keys.

Like the model 1000, the Timex 1500 uses an ordinary broadcast television receiver as the video monitor. The interface hardware is exactly

FIGURE 1-3. The Timex/Sinclair T/S-1500 keyboard.

the same, and in a subsequent chapter you will find that the interconnection is the same for both. There is no TV channel switch on the 1500, as there was on the 1000. The 1500 also uses either channel-2, or channel-3, but its selection is under keyboard control. Normally, the 1500 will send video to the television set on channel-2, but you can select channel-3 by pressing the "3" key immediately after the 1500 is turned on. According to the Timex 1500 manual, if a "3" appears on the screen when you change channels, then you just press "ENTER" to delete it.

Hardware interfacers are both good news and bad news on the 1500, especially if they already possess a certain amount of 1000 hardware that they would like to use on the 1500. The good news is that the same interface/peripheral connector is used on the 1500 as was on the 1000. The bad news is that it shares the back panel with the cassette, television and power supply connectors. This fact means that add-on accessories such as the *Memotech* devices do not easily mate with the 1500. In a later chapter we will see a "homebrew" connector extender that will permit you to use these devices. Also, you can use accessories such as the *Micro Development* accessories because they come equipped with a commercially-made extender cable.

The Timex 1000 included only 2K of internal RAM memory, with a 16K RAM available as an extra, bolt-on pack. The 1500, on the other hand, contains the 16K of RAM inside of the computer. Additional memory can be added, for a total of 64K. Be careful when ordering nonoriginal Timex memory modules for this application, however—especially the "homebrew" variety that are popular with some hobbiest users. Be sure that the memory module selected can be switched to a higher order of memory. Some products will want to start at OK, while others will start at 16K. There is a pin on the interface connector that is used to turn off the internal RAM, and this can be used by memory modules.

T/S-2068

The T/S-2068 (Fig. 1-4) is a giant leap above either the 1000 or the 1500: it is truly a sophisticated microcomputer complete with color capability. In addition to the interface connector on the rear apron of the 2068 (see Fig. 1-5), there is also a printed circuit card-edge connector on the front (under a cover—see Fig. 1-6) that accepts plug-in software cartridges. Obviously, some smart "interface hacker" will also figure out some way to use this connector for other applications than video games.

Physically, the TIMEX 2068 is considerably larger than the two predecessors. This computer is 15 inches wide, 7.5 inches deep and 2 inches high. It weighs approximately 4 pounds.

The interface connector on the rear apron is larger than the similar connector on the 1000 and 1500, and contains more functions. But a portion

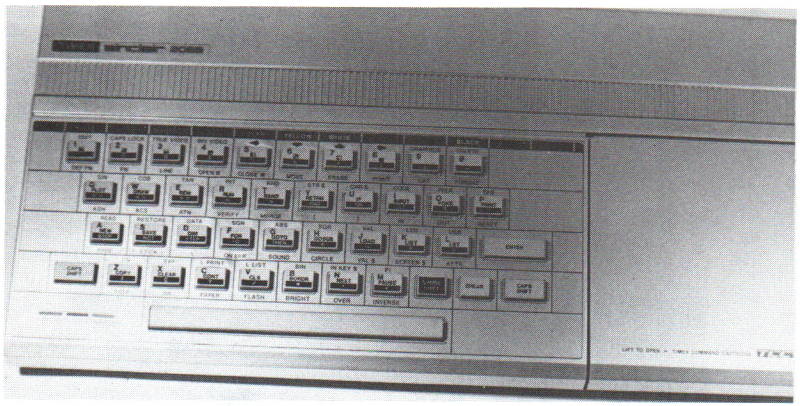


FIGURE 1-4. The T/S-2068 microcomputer.

of the connector is "T/S-1000 compatible," which means that many of the 1000/1500 accessories are usable on the 2068 as well. This expansion explains the use of the seemingly unusual "slotted-ends" connector on the old 1000. I don't know whether this was a "pre-planned product improvement" concept when the T/S-1000 was designed, or whether it is merely serendipity. Regardless, it is also very handy!

Two additional interface connectors that are likely to find greater use than apparently is their intent are the game connectors on the two sides. These are a standard form of 9-pin D-connector "Atari" style, and will be defined later on in this book.

The 2068 has one feature that I really missed on the old model 1000: an *on-off* switch. It is possible to turn the 2068 on and off from a rocker switch on the left-hand side apron.

Like the two earlier machines, the 2068 can use an ordinary broadcast television receiver as the video display monitor. With the 2068, however, Timex returns to the use of a mechanical switch to select either channel-2 or channel-3. This switch is located on the bottom side of the computer, in the upper right-hand corner. It comes from the factory set to channel-3, and can be changed to the channel-2 position by using a pen tip or similar instrument.

Added to the 2068 (not on the earlier models) is a composite video output that can be connected directly to a closed-circuit TV monitor, or one of

FIGURE 1-5. Interface connector on the rear apron of the 2068.

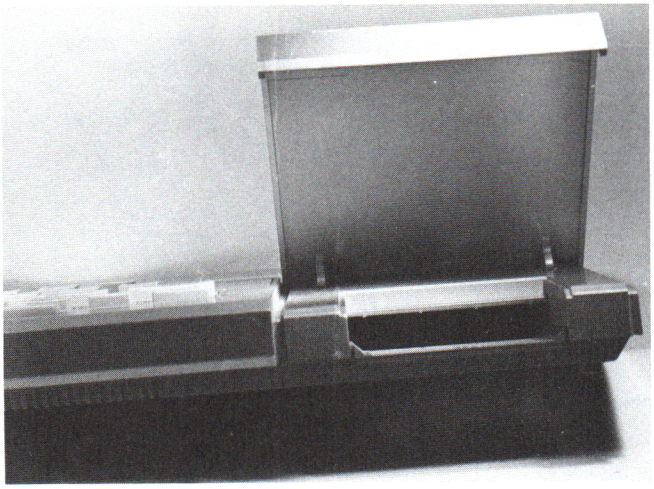


FIGURE 1-6. Printed circuit card-edge connector on the front of the 2068 accepts plug-in software cartridges.

the video computer monitors now on the market (which are really closed-circuit TV monitors, relabelled for computer use). This monitor permits high resolution displays, which are particularly useful in the graphics mode. You will find that such a monitor, while more expensive than using your regular television set, produces a more readable display because of the increased bandwidth. Because of both basic specifications and cost-cutting design practices, many ordinary television displays will not produce good, readable text on the screen when used as a computer monitor. The use of this monitor/video output should be limited to closed-circuit and/or computer monitors designed for that purpose. It is not recommended that you use homebrew modifications of existing television receivers for this application because there is a built-in safety problem! This particular modification became somewhat popular among the "underground" of users, not just of Timex but all similar computers. The modification can be done safely, but requires expert knowledge—and I don't mean miscellaneous "street knowledge" that purports to be expert—in order to be safe.

The keys on the 2068 are movable and tactile, like the 1500 keys, but are made of hard plastic. The author finds both 1500 and 2068 greatly improve 1000 membrane keys—but the keys on the 2068 are the best, in my opinion—that Timex had produced so far. I found the keyboard very easy to operate. The keyboard is laid out in the standard QWERTYUIOP sequence as on the old-fashioned typewriter keyboard. The "F" and "J" keys have a little raised dot that allows touch typists to tactilly find the "home" keys. There are 42 keys on the 2068 keyboard.

The memory arrangement in the 2068 is capable of extreme expansion. The Z80A uses a sixteen-bit address bus, so it can address 2^{16}, or 65,536 (i.e.

"64K") different memory locations. But with special design features, it is possible to address considerably more memory, even though only 64K at a time. The Timex 2068 comes from the factory equipped with 48K of user-addressable RAM, and 24K of internal ROM for a total of 72K of memory. The ROM is broken into two portions: 16K of fixed ROM and 8K of additional bank-selected ROM. The ROM contains the improved BASIC interpreter. One use of the bank-switched 8K portion of ROM is to allow expanded memory size for the 2.5-inch "Command Cartridges" that insert into the connector on the front panel.

The 2068 is also equipped with a loudspeaker which can be operated under program control to provide four different "voices." The SOUND command allows the programmer to specify tone duration, amplitude (i.e. intensity level) and duration, along with certain characteristics of the tone envelope; this command is used with three of the voices. The fourth voice is available under the BEEP command, and allows selection of duration plus 130 "semitones."

There are four different display modes on the 2068 device. Mode-1 is the same 22-row, 32-column (i.e. characters) format as used on the 1000 and 1500. Mode-1 uses 16 character-space graphic blocks for 256 pixels per line, with 176 lines of pixels from top of the screen to bottom.

Display Mode-2 is similar to Mode-1, except that it permits 64-columns (i.e. 64 characters per line). The graphics format in this mode is 512 pixels horizontally, with the same 176 pixel lines vertically.

Display Mode-3 is a second "Mode-1", and can be switched in and out under program control. One reviewer (*Radio-Electronics* magazine) claimed that animation can be produced by rapidly switching between modes 1 and 3.

Display Mode-4 is similar to Mode-1 except that each graphics character space contains eight rows, each consisting of 8 pixels.

The "Mad" Computer—A Discussion for the Beginner and Novice

The rest of this book is dedicated to discussions of the Z80A microprocessor chip—information that is needed for proper interfacing and machine language programming—and certain specific information regarding the Timex computers in particular. Before proceeding, however, we should deal with a few matters of concept and vocabulary. In the next chapter, we will describe certain computer terms, such as *microcomputer*, *minicomputer* and so forth, and discuss the operation of any elementary programmable digital computer. We will use a "universal" computer which we shall call the *Mythical Analytical Computer*, or MAD. It is a hypothetical generic computer that will permit you to relate to the text that follows and describes the Z80 microprocessor in particular. If you are already familiar with the elementary concepts of computer operation, then you may successfully skip Chapter 2.

CHAPTER 2

INTRODUCTION TO PROCESSORS AND PROCESSING

One of the most frequent questions asked by the new microcomputer owner is: "What will it do?" This question is so exasperating because it has too many *good* answers. Indeed, what *is* the role of a personal microcomputer? For that matter, what is a microcomputer?

At one time, definitions were simpler. As a freshman engineering student at Old Dominion University (Norfolk, Va.), I was allowed to use an IBM 1601-1620 machine; *that* was a computer! There was no doubt in anyone's mind about that machine's identity; it took up an entire room on the second floor of Kaufman Hall (the engineering school's building). But, today, an engineering student can sit at a small desk with a TRS-*80* or APPLE II (complete with video CRT display, printer and two disc drives) that has more computing power than that old 1601! In fact, many engineering school students find that the cost of the typical small system is affordable and now they own their own computer. The student can almost have more computing power in a T/S-1000 than we had back then in the entire School of Engineering! The cost of the modern microcomputer is less than one-tenth of what one of the lesser machines of only a decade ago cost—not counting the fact that 1971 dollars were bigger than today's dollars—and T/S-1000 costs under $100.

Before attempting to define the role of the microcomputer, let's first try to define the microcomputer. Terminology tends to become sloppy both from

our own laziness and from the fact that once genuine distinctives have become blurred as the state-of-the-art advances, terminology in the computer field is often "O.B.E."—overcome by events. For example, consider the terms *microcomputer* and *minicomputer*. Laziness tends to make some of us use these terms interchangeably, while modern single-chip computers (e.g., the Intel 8048 device) tend to make such usage seem reasonable. But, for our purposes, we require sharply focused meanings for these two terms and others: *minicomputer, microcomputer*, *single-chip computer*, *single-board computer*, and *mainframe computer*.

MICROPROCESSOR

The *microprocessor* is a large scale integration (LSI) integrated circuit that contains the *central processing unit* (CPU) of a programmable digital computer. The CPU section of a computer contains the *arithmetic logic unit* (ALU) that performs the basic computational and logical operations of the computer. The CPU also houses the *control logic* section (which performs housekeeping functions) and may or may not have several *registers* for the temporary storage of data. All CPUs have at least one temporary storage register called the *accumulator*, or *A-register*. The principal attribute of a microprocessor is that it will execute instructions sequentially. These instructions are stored in coded binary form in an external *memory*.

MICROCOMPUTER

A *microcomputer* is a full-fledged programmable digital computer that is built around a microprocessor "chip" (i.e., integrated circuit); the microprocessor acts as the CPU for the computer. In addition to the microprocessor chip, the microcomputer typically will have some additional chips; the number varying from two to hundreds depending upon the design and the application. These external chips may provide such functions as memory (both temporary and permanent), input/output (I/O) and other functions. The microcomputer may be as simple as a T/S-1000, or, as complex as a thirty-board professional machine with all the electronic data processing "goodies."

SINGLE-CHIP COMPUTER

For several years we had no excuse for interchanging the terms *microprocessor* and *microcomputer*; a μP was an LSI chip and μC was a computing machine. But the 8048 and similar devices began the process of dissolving the previously

well-defined boundaries because it was both an LSI IC *and* a computer. A typical single-chip computer may have a CPU section, two types of internal memory (temporary and long-term permanent storage) and at least two I/O ports. Some machines are even more complex.

The single-chip computer does, however, require some external components before it can do work. By definition, the microcomputer already has at least the minimum of components needed to perform a job.

SINGLE-BOARD COMPUTERS

The *single-board computer* is a programmable digital computer, complete with input and output "peripherals," on a single printed circuit board. Popular 6502-based examples are the KIM-1, SYM-1 and AIM-65 machines. The single-board computer might have either a microprocessor, or, a single-chip computer at its heart. The T/S-1000 is technically a single-board computer even though it comes packaged in a plastic cabinet.

The peripherals on a single-board computer are usually of the most primitive kind (T/S-1000 is a notable exception) consisting of seven-segment LED numerical displays and hexadecimal keypads reminiscent of those on a handheld calculator or Touchtone[R] telephone. The typical display is capable of displaying only hexadecimal numerical characters, and even these are a little optimistic because of the form constraints of using seven-segment LED display devices.

Most single-board computers have at least one interface connector that allows either expansion of the computer, or, interfacing into a system or instrument design.

The manufacturers of SBCs, such as the early KIM-1 and others, probably did not envision their wide application as a small-scale development system. These computers were primarily touted as trainers, that is for use in teaching microcomputer technology. But, for simple projects, such computers also work well as a mini development system! More than a few SBC trainers have been used to develop a microcomputer-based product, only to wind up being specified as a "component" in the production version. In still other cases, the commercially available SBC was used as a component in prototype systems, and then, in the production version, a special SBC (lower cost) was either bought or built.

MINICOMPUTER

The *minicomputer* predates the microcomputer and were originally little more than scaled-down versions of larger data processing machines. The Digital Equipment Corporation (DEC) PDP-8 and PDP-11 machines are

examples of "minis." The minicomputer will use a variety of small-scale (SSI), medium-scale (MSI) and large-scale integration (LSI) chips.

Minicomputers have traditionally been more powerful than microcomputers. They had, for example, longer length binary data words (12 to 32 bits instead of 4 or 8 bits found in micros!), and operated at faster speeds (6 to 12 mHz instead of 1 to 3 mHz). But this is an area of fading distinctives! Digital Equipment Corporation, for example, offers the LSI-11 microcomputer that acts like a mini. Similarly, 16-bit microcomputers are available, as are 6 mHz devices. It is sometimes difficult to draw the line of demarcation when a microcomputer is in the same size cabinet as a minicomputer, and minicomputers can be bought in desktop configurations!

MAINFRAME COMPUTERS

The large computer that comes to mind when most people think of computers is the *mainframe computer*. These machines are the computers used in large scale data processing departments. Microcomputerists who have an elitist mentality sometimes call mainframe computers "dinosaurs." But, unlike their reptilian namesakes, these dinosaurs show no signs of extinction and are, in fact, an evolving species. The IBM 370 is an example of a mainframe computer.

ADVANTAGES OF MICROCOMPUTERS

That microcomputers have certain advantages is attested by the fact that so many are sold! But what are these advantages, and how are they conferred?

The most obvious advantage of the microcomputer is *reduced size*; compared with dinosaurs, microcomputers are mere lizards! An eight-bit microcomputer with 64K bytes of memory can easily fit inside of a tabletop cabinet.

The LSI microcomputer chip is generally more complex than a discrete components circuit that does the same job. The interconnections between circuit elements, however, are much shorter (micrometers instead of millimeters). Input capacitances are thereby made lower. The MOS technology used in most of these ICs produces very low current drain, hence the overall reduced heating. While a minicomputer may require a pair of 100 c.f.m. blowers to keep the temperature within specifications, a microcomputer may be able to use a single 40 c.f.m. muffin fan, or, no fan at all!

Another advantage of the LSI circuit is reduced component count. Although this advantage relates directly to reduced size, it also affects reliability. If the LSI IC is just as reliable as any other IC (and so it seems), then the overall reliability of the circuit is increased dramatically. Even if the

chip reliability is lower than lesser ICs, we would still achieve superior reliability due to fewer interconnections on the printed circuit board, specially if IC sockets are used. Some of the most invideous troubleshooting problems result from defective IC sockets!

MICROCOMPUTER INTERFACING

The design of any device or system in which a microcomputer or microprocessor is used is the art of *defining* the operation of the system or device, *selecting the components* for the device or system, *matching* and *integrating* those components (if necessary), and *constructing* the device or system. These activities are known collectively as *interfacing*.

But let's get down to a more basic level. Most readers of this book are technically-minded people with some knowledge of electronics and computer technology. For most readers, therefore, interfacing consists of selecting and matching components, and then connecting them into a circuit that does a specific job. These matters are addressed in other chapters.

MICROCOMPUTERS IN INSTRUMENT AND SYSTEM DESIGN

Designers in the past had to use analog electronic circuits, electromechanical relays (which sometimes lead to a maintenance nightmare) and other devices in order to design instruments, process controllers and the like. These circuit techniques had their limitations and produced some irritating results—factors like thermal drift loomed large in some of these circuits. In addition, the design was cast in cement once the final circuit was worked out. Frequently, relatively subtle changes in a specification or requirement produced astonishing changes in the configuration of the instrument; analog circuits are not easily adaptable to new situations in many cases. But, with the advent of the microcomputer, we gain the advantage of flexibility and solve some of the more vexing problems encountered in analog circuit design. The memory of the computer tells it what to do, and that can be changed relatively easily. We can, for example, store program code in a *read only memory*, or ROM, which is an integrated circuit memory. If a change is needed, then the software can be modified and a new ROM installed. If the microcomputer was configured in an intelligent manner, then it is possible to redesign only certain interface cards (or none at all) to make a new system configuration! An engineer of my acquaintance built an anode heat computer for medical X-ray machines. A microprocessor would compute the heating of the anode as the X-ray tube operated, and would sound a warning if the limit of safety was exceeded—thus saving the hospital the cost of a $10,000 X-ray tube! But different X-ray

machines require different interfacing techniques, a problem that previously had meant a new circuit design for each machine. But, by intelligent engineering planning, the anode heat computer could be built with a single interface card that married the "universal" portion of the instrument with each brand of X-ray machine. Thus, the company could configure the instrument uniquely for all customers at a minimum of cost.

Another instrument that indicates the universality of the microcomputer is a certain cardiac output computer. This medical device is used by intensive care physicians to determine the blood pumping capability of the heart in *liters per minute*. A "bolus" of iced or room temperature saline solution is injected into the patient at the "input" end of the right side of the heart (the heart contains two pumps, right side and left side, with the right side output feeding the left side input via the lungs). The temperature at the output end of the right side is monitored, and the time integral of temperature determined. This integral, together with some constants, is massaged by the computer to calculate the cardiac output. These machines come in two versions, research and clinical. The researcher will take time to enter certain constants (that depend upon the catheter used to inject saline, temperature and other factors) and will be more vigorous in following the correct procedure. But, in the clinical setting technique suffers to the needs of caring for the patient and the result is a perception of "machine error," which is actually operator error. So, to combat this problem, the manufacturer offers two machines. One is a research instrument and is equipped with front panel controls that allow the operator to select a wide range of options. The other, a clinical model, allows no options to the operator and is a "plug and chug" model. The interesting thing about these instruments is that they are *identical* on the inside! All that is different is the front panel and the position of an on-board switch! The manufacturer's program initially interrogates a switch to see if it is open or closed. If it is open, then it "reads" the keyboard to obtain the constants. If, on the other hand, it is closed, then the program branches to a subprogram that assumes certain predetermined constants that are loaded on the buyer's prescription at the time the instrument is delivered. The cost savings of using a single design for both instruments are substantial!

MICROPROCESSOR FUNDAMENTALS

The microprocessor chip literally revolutionized the electronics industry. Although initially thought of as either a small logic controller, or, as a data processing machine (depending upon your perspective and the first chip you saw), the microcprocessor blossomed into a major force with hundreds or even thousands of applications in less than a decade.

What is a "microprocessor"? How does it relate to a "microcomputer"? In this book we will explore these questions, and hopefully give you a good

grounding in computer technology basics. But first, we will study computers in general by describing a "typical" programmable digital computer in block diagram form. In chapters to follow we will study the Z80 device which forms the heart of the T/S-1000 computer.

Mythical Analytical Device (MAD)

Rather than mold our discussion around any one manufacturer's product, let's make up one that is significantly general to cover a large number of actual devices. Our "thought-computer" will be nicknamed the Mythical Analytical Device, or MAD, because the acronym MAD adequately describes both the emotional state of programmers (whose frustrations mount geometrically with each passing "bomb-out") and the mental health state of computer sciences freaks (who are often seen wandering aimlessly through university corridors muttering the arcane glossolalian prayers of their modern religion: "Hail, microprocessor, from whom all bits and bytes emmanate. . .").

Figure 2-1 shows the block diagram of MAD. Let's first describe the bits and (*errrr* that should be "parts") and pieces, and then describe a typical program operation.

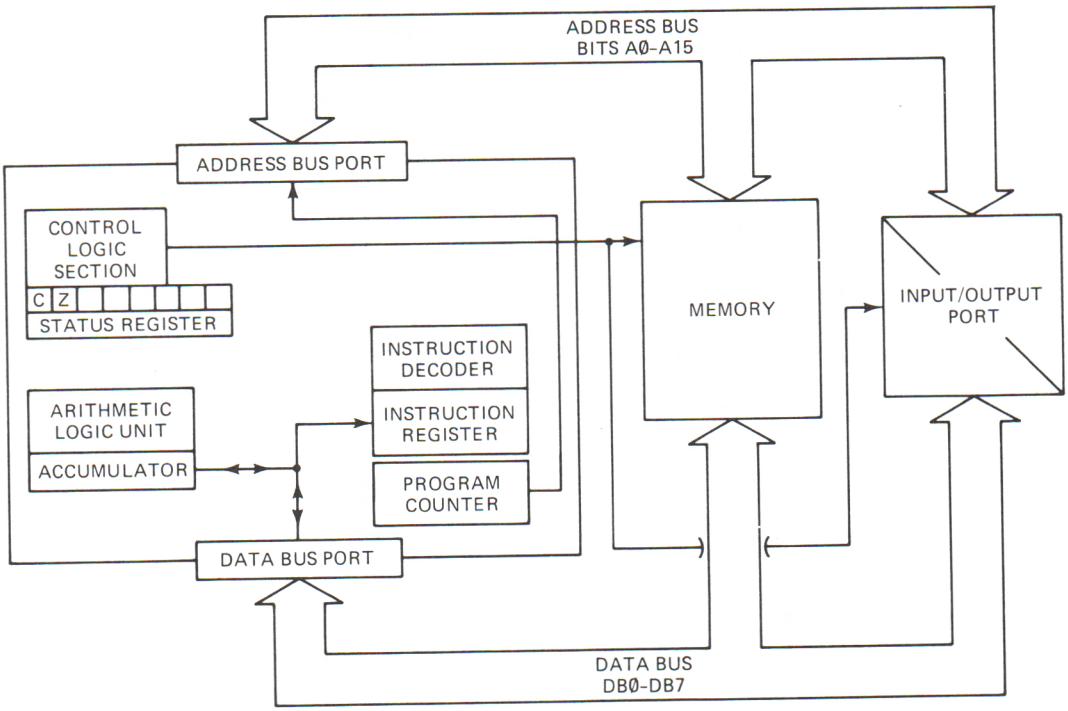


FIGURE 2-1. Block diagram for MAD (Mythical Analytic Device) computer.

MAD, like any programmable digital computer, has three main parts: *central processing unit* (CPU), *memory*, and *input/output* (I/O). There are certainly other functions in some specific machines, but many of these are either special applications of these main groups, or, are too unique to be described in a general machine.

The *central processing unit*, or CPU, controls the operation of the entire computer. It consists of several necessary subsections which will be described in greater detail below.

Memory can be viewed as an array of "cubbyholes" such as those used by postal workers (Fig. 2-2) to sort mail. Each cubbyhole represents a specific *address* on the letter carrier's route. An address in the array can be uniquely specified (identifying only *one* location) by designating the *row* and *column* in which the cubbyhole is found. If we want to specify the memory location (i.e., cubbyhole) at row 3 and column 2, then we could create a *row X column* "address number" which, in this example, would be 32.

Each cubbyhole represents a unique location in which to store mail. In the computer, the memory location stores a binary word of information. In an eight-bit computer, each location will store a single eight-bit binary word.

The different types of memory devices are disscussed in another chapter, so that information will not be given here; to the CPU is doesn't matter much whether memory at any given location is RAM or ROM.

There are three lines of communication between the memory and the CPU: *address bus*, *data bus* and *control logic signals*. These avenues of communication control the interaction between memory and I/O, on the one

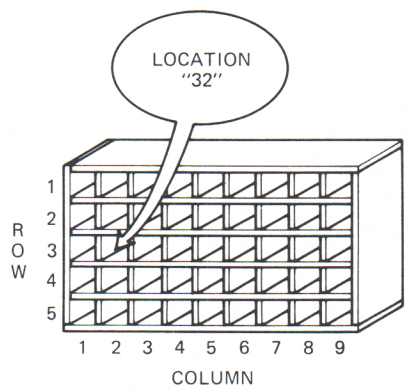


FIGURE 2-2. "Cubbyhole" analogy for memory.

hand, and the CPU on the other. Ultimately, therefore, they also control the functioning of the entire computer.

The *address bus* (bits A0 through A15 in Fig. 2-1) communicates to the memory bank the address of the exact memory location being called by the

CPU, regardless of whether a read or write operation is taking place. The address bus consists of parallel data lines, one for each bit of the binary word that is used to specify the address location. In most eight-bit microcomputers, for example, the address bus consists of sixteen bits. A 16-bit address bus can uniquely specify 2^{16}, or 65, 536 different locations. This size is called "64K" in computerese, not "65K" as one might expect if reason was operative. It seems that "*k*" represents the metric prefix *kilo*, which denotes "1,000." Since 2^{10} is 1,024, however, computerists long ago decided that kilo would be 1,024 not 1,000. The "big-k" (1,024) is represented with an uppercase "*K*" rather than "*k*" used for real *kilo*.

Be aware that the size of memory which can be addressed *doubles* for every bit added to the length of the address bus. Hence, adding one bit to our sixteen-bit address bus creates a seventeen-bit address bus which can designate up to 128K locations. Some "8-bit" machines which have 16-bit address buses can be made to look like bigger machines by certain tactics that make a longer pseudo-address bus. In those machines, several 64K memory banks are used to simulate continuously addressable 128K, 256K or 512K memories.

The *data bus* is the communications channel over which data travels between the main register (called the *accumulator* or *A-register*) in the CPU and the memory. The data bus also carries data to and from the various input and/or output ports. If the CPU wants to "read" the data stored in a particular memory location, then that data is passed from the memory location over the data bus to the accumulator register in the CPU. Memory write operations are exactly the opposite direction, but otherwise the same.

The size of the data bus is usually cited as the "size" of the computer. An "8-bit" microprocessor/microcomputer, therefore, is one that has an 8-bit data bus; a 16-bit microcomputer will have a 16-bit data bus. Do not be confused by salesmen such as the bozo who told me his 6502-based machine (8-bit data bus) was "in reality" a 16-bit machine because it had a 16-bit address bus!

The last memory signal is the *control logic* or timing signal. These are one or more binary logic signals that tell memory *if* it is being addressed, and whether the request is a read or write operation. The details of control logic signals differ between microprocessor chips, so only those of the Z80 will be discussed in this book. (For Z80 signals see *Z80 Users Manual*, by J. J. Carr, Reston, Va: Reston Publishing Co., 1980.)

The *input/output* (I/O) section is the means by which the CPU communicates with the outside world. An input port will bring data in from the outside world and then pass it over the data bus to the CPU where it is stored in the accumulator. An output port reverses that data flow direction.

In some machines, there are separate I/O instructions that are distinct from memory instructions. The Z80 is one such machine. The Z80 will pass the port address over the lower eight bits of the sixteen-bit address bus (8-bit I/O

address used in the Z80 can uniquely address up to 256 different ports). In other machines, such as the 6502, there are no distinct I/O instructions. In those machines, the I/O components are treated as memory locations; this technique is called *memory-mapping* or *memory-mapped I/O*. Input and output operations then become memory-read and memory-write operations, respectively.

Central Processing Unit (CPU)

The CPU is literally the heart and brains of any programmable digital computer, including MAD. Although there are some different "whistles and bells" features in certain machines, all will have the features shown in our MAD computer (Fig. 2-1). The principal subsections of the CPU include (at least) the following: *accumulator* or *A-register, arithmetic-logic unit* (ALU), *program counter* (PC), *instruction register*, *status register* and *control logic section*.

The accumulator is the main register in the CPU, and will have the same bit-length as the data bus. All instructions executed by the CPU involves data in the accumulator, unless otherwise specified in the description of that instruction. Therefore, an "ADD" instruction causes an arithmetic addition of the data cited by the instruction to the *contents of the accumulator*.

Although there are often other registers in the CPU, the accumulator is the main register. The main purpose of the accumulator is the *temporary* storage of data being operated on by the instruction being executed. Note that data transfers to and from the accumulator are non-destructive. In other words, data "transfers" are not really transfers at all, but are, instead, "copying" operations. Suppose, for example, the hexadecimal number $8F_{16}$ is stored in the accumulator when an instruction is encountered requiring that the contents of the accumulator be stored at memory location $A008_{16}$. After the instruction is executed, we will find $8F_{16}$ both in memory location A008 *and* in the accumulator. If we have the opposite operation (i.e., transfer contents of accumulator to location $A008_{16}$), then we will see the same situation; after the transfer, the data will be in both locations. Since the accumulator contents change every time an instruction is executed, we will have to use such transfers to hold critical data some place in memory.

The arithmetic-logic unit (ALU) contains the circuitry that performs the arithmetic operations of addition and (sometimes) subtraction, plus the logical operations of AND, OR, and XOR.

The program counter (PC) contains the address of the next instruction to be executed. The secret to the success of a programmable digital computer is its ability to *fetch and execute instructions sequentially*. Normally, the PC will increment appropriately (1, 2, 3 or 4) while executing each instruction: 1 for 1-byte instructions, 2 for 2-byte instructions, et cetera. For example, the instruction "LDA,n" is a 6502 instruction mnemonic that loads the accumu-

lator with the number "n." In a program, we will find the code for LDA,n followed by "n."

Location	Code	Mnemonic
0100		LDA,n
0101	"n"	"n"
0102	(next instruction)	

At the beginning of this operation, PC = 0100, but after execution it will be PC = 0102 because LDA,n is a two-byte instruction.

There are other ways to modify the program counter. Executing any form of JUMP instruction, for example, modifies the contents of PC to contain the address of the "jumped-to" location. Another way to modify the PC contents is to activate the *reset* line. The computer sees reset as a hard-wired JUMP to location FFFA.

The *instruction register* is the temporary storage location for instruction codes stored in memory. When the instruction is fetched from memory by the CPU, it will reside in the instruction register until the next instruction is fetched.

The *instruction decoder* is a logic circuit that reads the instruction register contents, and then carries out the intended operation.

The *control logic section* takes care of housekeeping chores within the CPU, and issues or responds to control signals from the outside world. These signals are not universally defined, but control such functions as memory requests, I/O requests, read/write signaling, interrupts and so forth.

The *status register*, also sometimes called *status flags*, is used to indicate to the program and (sometimes) to the outside world the status of the CPU at any given instant. Each bit of the status register represents a different function. Different microprocessor chips use slightly different status register architectures, but all will have a *carry flag* (C) to indicate when an instruction execution caused a "carry," and a *zero flag* (Z) that indicates when an arithmetic or logic instruction resulted in zero or nonzero in the accumulator (typically, Z = 1 when result is zero).

We have now developed the CPU for our MAD computer. This discussion in general terms also describes a typical microprocessor chip; a microprocessor (as opposed to a single-chip computer) is essentially the CPU portion of a MAD.

Operation of MAD

A programmable digital computer such as MAD operates by sequentially fetching, decoding and then executing instructions stored in memory. These instructions are stored in the form of *binary numbers*. In some early machines

there were two memories, one each for program instructions and data. The modern method, however, uses the same memory for both data and instructions.

How, one might ask, does the dumb computer know whether the binary number stored in any particular location is an instruction, data or an alphanumeric character representation (e.g., ASCII or BAUDOT codes)? The answer to this important question is the key to the operation of MAD: *cycles*.

The MAD operates in cycles. A computer will have at least two cycles: *instruction fetch* and *execution*, and in some machines these cycles are sub-cycles. The details differ even though general scenarios are similar.

Instructions are stored in memory as binary numbers called *operation codes* (op-codes, for short). During the instruction fetch cycle, an op-code will be retrieved from the memory location specified by the program counter and stuffed into the instruction register. The CPU assumes that the programmer was smart enough to arrange things such that an op-code will be stored at that location when the PC increments to that address.

During the first cycle, an instruction is fetched and stored in the instruction register. During the second cycle, the instruction decoder will read the IR, and then carry out the indicated operation. When these two cycles are completed, an instruction will have been fetched and executed, the program counter incremented to reflect the memory location that will contain the next instruction, and the CPU made ready for the next instruction. The CPU will then enter the next instruction fetch cycle and the process repeats itself.

The above process continues over and over again as long as the MAD is working. Each step is synchronized by a train of clock pulses so that events remain rational.

From the above description you might be able to glean a truth concerning what a computer can or cannot do. The CPU can shift data around, perform logical operations (e.g., AND, OR, XOR), add two N-bit numbers (sometimes subtract as well), all in accordance with a limited repertoire of binary word instructions. These chores are performed *sequentially* through a series of discrete steps. The secret to whether a problem is amenable to computer solution depends upon whether a plan of action (called an *algorithm*) can be written that will lead to a solution by a sequentially executed series of steps. Most practical instrumentation, control or data processing chores can be so solved—a factor which accounts for the meteoric rise of the microprocessor. A field of endeavor that studies sequential solutions to practical (and some not so practical) problems is called *numerical methods*.

The MAD computer is merely a hypothetical construct used as a teaching aid. Let's examine a *real* microprocessor—the Z80.

CHAPTER 3

SETTING UP THE TIMEX COMPUTERS

The Timex T/S-1000, 1500, 2068 and Sinclair ZX-81 microcomputers are designed to be operated with simple external peripherals. The display device, for example, is any ordinary black-and-white television receiver. Of course, we can use the color television also, but the T/S-1000 will not display in color. The mass data storage is in an ordinary audio cassette recorder. As we discussed in Chapter 1, we should have one of reasonable quality, and that includes a counter mechanism for relocating programs on the tape. The cassette recorder must also have provisions for external microphone and earphone connections (the standard 3.5 mm jacks used on Japanese transistor radios are used on the T/S-1000).

This book was originally planned for the Timex T/S-1000 user before the 1500 and 2068 models were produced. Thus, the connection instructions that follow use the language of T/S-1000. But the connections are the same for the other two computers, except for the location of the connectors. On the 1500 and 2068 computers, the connectors are along the rear apron instead of the left-hand side of the computer. Otherwise, connection is the same. (*Note:* There are power supply differences, so only use the power pack supplied with the specific computer!)

When you unpack the T/S-1000, you will notice that several pieces come with the computer: *power supply and cord, TV output cord, TV*

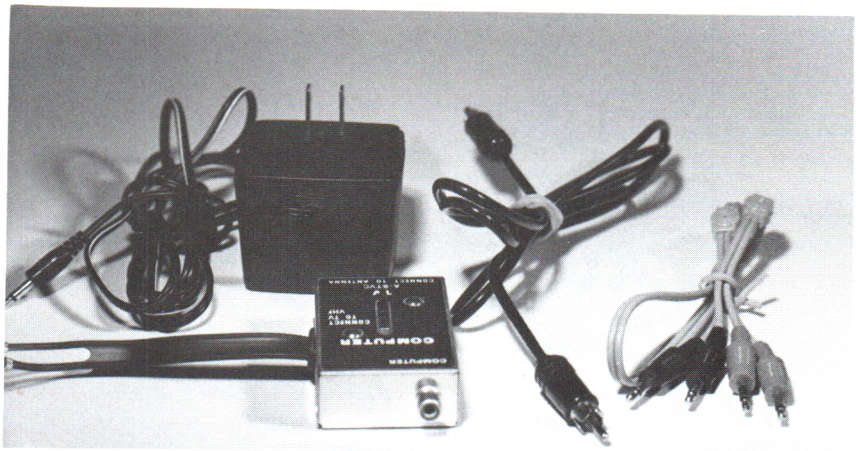


FIGURE 3-1. Accessories for T/S-1000.

antenna switch box, and a dual *cassette lead* (see Fig. 3-1). These accessories are used as follows:

1. *Power Supply and Cord.* This device is much like the power packs used on calculators and other small appliances. It produces a DC potential of +9.75V DC at a current of 650 milliamperes (0.65 amperes). This supply is built inside of the housing that contains the AC power line plug, and is connected to the T/S-1000 via a low-voltage cable.
2. *TV Output Cord.* This cord is a shielded coaxial cable with an RCA-style male phono plug on each end. It is used to connect the RF/video output of the T/S-1000 to the TV antenna switch box.
3. *TV Antenna Switch Box.* The T/S-1000 is designed to connect directly to the antenna input of the television receiver. The switch box (see Fig. 3-2) performs two functions. First, it converts the output impedance of the T/S-1000 to the 300-ohm input impedance of the television receiver. Secondly, it also serves to permit us to switch back and forth between the computer and the antenna, which could be important if you don't want to tie up the television set all of the time for the computer.
4. *Cassette Lead.* The cassette outputs of the T/S-1000 match the 3.5 mm microphone and earphone jacks on most transistor cassette players. The Timex T/S-1000 comes with a dual cable that consists of two shielded cables joined together. In order to keep the ends separate and easily identified, the plugs are color coded. Just make sure that that same color as is attached to the "MIC" output of the

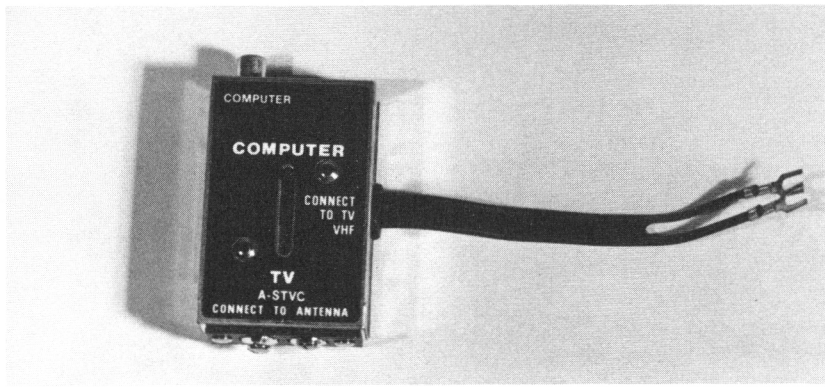


FIGURE 3-2. Antenna switch box.

T/S-1000 is also used on the "MIC" input of the cassette player; the same advice also holds for the "EAR" jacks.

Figure 3-3 shows the sideview of the T/S-1000 microcomputer. There are four jacks on this side: TV, EAR, MIC and 9V DC. The TV jack is the RF output to the television receiver. The signal from this jack is an RF signal on TV channels 2 or 3, depending upon the setting of a switch on the bottom of the T/S-1000, and is modulated with the video signal carrying the display information. The EAR jack is used to connect to the earphone jack of the cassette recorder. Audio signals recovered from the tape are output from the recorder through the earphone jack and are therefore input to the computer through the EAR jack shown here. *Note:* Some people find that it is

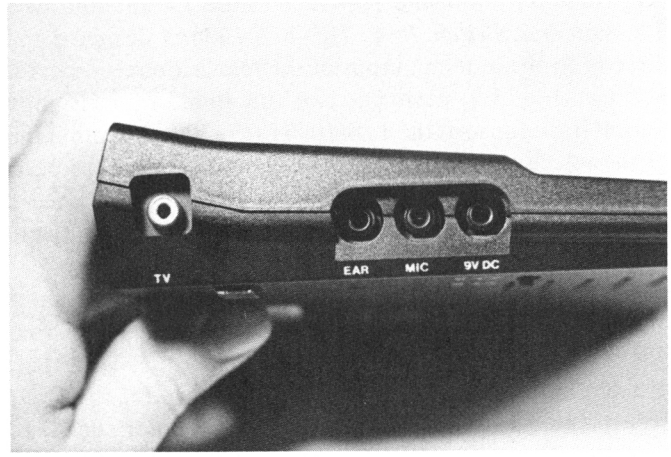


FIGURE 3-3. Sideview of T/S-1000 showing input/output and power supply connectors.

disconcerting to try locating programs because the earphone jack cuts off the cassette player speaker. Some will therefore short out the jack inside the cassette player so that the speaker remains active even though signal is being output to the EAR jack. This job is relatively simple for an experienced technician, but should not be attempted unless you are familiar with cassette tape recorder service. Also, there are a few models in which this cannot be done without endangering the cassette player.

The MIC connector is the T/S-1000 audio output. Signals sent from the computer to the tape player exit via this jack. Like the EAR jack, this connector is the standard 3.5 mm subminiature phone jack.

The 9V DC jack, also a 3.5 mm subminiature phone jack, is for the unregulated power supply. The output of the power supply is attached to this jack. There is an internal 5 volt DC three-terminal regulator that converts the 9 volt unregulated source to +5 volts regulated, as required by the Z80 and other digital integrated circuits used in the T/S-1000.

Figure 3-4 shows the bottom of the T/S-1000 microcomputer. The channel selector switch shows opposite-pointing arrows that will determine whether the RF output is on TV channel-2 or channel-3. It is very rare for the FCC to assign adjacent channels to different broadcasters in the same market, so one of these ought to be useful to you. In some cases (frequently, I suspect), there might be a channel-4 station. Theoretically, we should be able to use either ch-2 or ch-3 for the computer, but best results will be obtained if channnel-2 is used. The problem is that many TV receivers, especially the low cost B&W jobs that most people will use with the T/S-1000, do not reject

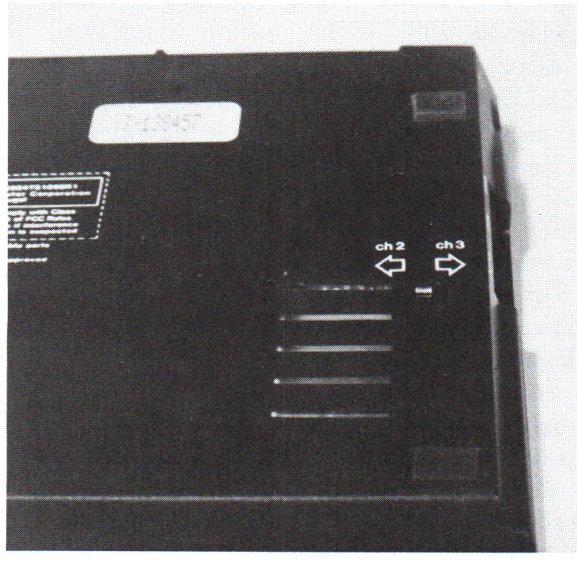


FIGURE 3-4. Channel selector switch.

adjacent channel interference as well as they might, so there is a large potential for interference. In some areas, like mine for example, we have ch-2 and ch-4 only 35 miles apart, so ch-3 is the only usable T/S-1000 channel and we have to ignore the herringbone interference pattern. This problem is not so acute as to render the T/S-1000 difficult to use, however.

Another source of interference might be the T/S-1000 itself. Digital computers use, of course, digital circuits which operate on the basis of fast rise-time squarewave pulses. The clock in the T/S-1000 is a 3.5 mHz generator of such pulses. These pulses can interfere with television reception at close range, and might show up as a noise pattern on the TV video screen. The cure seems to be placing the T/S-1000 as far from the TV as the interface cord permits, and to place the television receiver a few inches higher than the T/S-1000. A small platform made from scrap lumber should prove more than sufficient if you have no other appliance to do the trick. In my own case, I used a small stack of unread books from my library to support the TV during computing sessions.

The T/S-1000 is not intended to be input directly to a closed-circuit TV monitor, such as used with closed-circuit TV systems and other forms of microcomputer. These monitors almost always have a wider bandwidth in the video amplifiers than is possible with broadcast-grade television receivers. Such instruments are little more than a television receiver with the video amplifiers improved and the IF/RF (tuner) stages removed. The implication of using a wider bandwidth is that the display is crisper and more easily read than on a television. The use of the television by the T/S-1000 is one of those cost trade-offs that make sense on a low cost computer, but can be improved upon if you want to spend the money. A monitor TV will cost from $150 to $300 in monochrome (either B&W, which produces either black letter on white background, or, white letters on black background, or, in green phosphors which produces green letters on the black background). If you are willing to void the warranty, or, if the warranty on your unit is expired, then it is possible to modify the T/S-1000 to provide a direct-video output that can drive a monitor as described above. There are three terminals going into the on-board RF modulator inside of the T/S-1000: common (ground),+5 volts and video. Of course, the RF comes out the RCA phono jack on the other side of the modulator subassembly. If you place a piece of coaxial cable such that the shield is attached to ground or to the modulator's metal chassis, and the center conductor is attached to the modulator's video input, then this cable can be used to directly drive the video monitor. Of course, you will have to somehow attach either an RCA-style phono jack, or, a BNC jack to the T/S-1000 case if you want to be easily disconnectable. Otherwise, just run the coax directly through a small hole in the cabinet (which you create with a small drill or soldering iron tip) to the video monitor. Perhaps the best form of coaxial cable used for this purpose is the RG-174 type, which is very tiny

compared with RG-59 (which is normally used for closed-circuit TV applications).

There is one problem which is not often mentioned in the microcomputer literature, but none the less occassionally plagues users of such equipment. The RF video signals is a miniature transmitter, and can therefore interfere with neighboring TV receivers under the right conditions. The TV antenna switch box is never totally unilateral—meaning that it transmits signals in only one direction—so there is a possibility that signals meant for your TV are radiated into the neighborhood. This problem is exacerbated by the use of high gain TV antennas, which can be identified as the ones having the most elements (i.e. cross-arms). Such antennas essentially amplify (bad word technically, but a good analogy) the minute video signal that comes through the switch box. If this problem is found on your system, then it can be solved by disconnecting the outside antenna (or the TV rabbit ears) from the switch box while using the computer. I do not expect this problem to manifest itself widely, but if you find that your neighbors can pick up your video signal, thereby see what you are programming, then it might be prudent to take the above mentioned precaution.

TV RF modulators for products like the T/S-1000 have to meet FCC regulations concerning such radiation before they can be sold in the USA. Therefore, it is rare for a problem to occur, but defects and anomolies do occur, so be aware of the potential for problems. A reasonably good test is to set up your computer, display some sort of silly message on the screen, and then go to your neighbors house and see if you can find the message on their TV on ch-2 or ch-3 (as you have set up your T/S-1000). The best neighbor to pick is one whose home is directly in the path of your outdoor TV antenna. Such radiations are, incidentally, quite illegal under FCC Rules and Regulations.

Figure 3-5 shows the correct method for connecting the T/S-1000 to the television receiver and to the cassette player. The connections are reasonably straight forward, and should require but little elaboration:

1. *Antenna switch box.* Disconnect the twin-lead antenna wire attached to the television receiver, and reconnect this wire to the twin slot-screw terminals on the switch box marked "Connect to Antenna." Connect the spade lugs on the twin lead coming from the switch box to the antenna terminals on the television from whence you just removed the antenna lead. Connect the TV cable (identified as a cable about 4-ft long, with male RCA-style phono plugs on both ends) to the female jack on the switch box marked "Computer." Connect the other end of this cable to the "TV" output on the left-hand side of the T/S-1000.

   Note: There is one possible situation where the connection

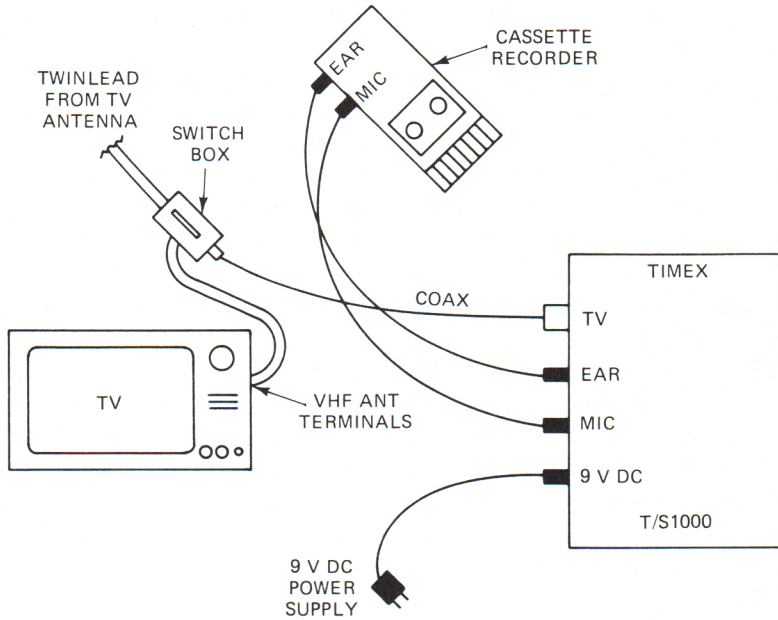


FIGURE 3-5. Connecting the T/S-1000.

between the switch box will not be as above. If you have a coaxial cable from either an antenna or from your local subscription cable—TV system, then you will require transformers to convert between the round coaxial cable and the flat twin-lead for which the switch box was designed. A 75-ohm-coax-to-300-ohm-twin-lead transformer is needed between the antenna line and the switch box. In most cases, television receivers are equipped or 300-ohm twin-lead, so this transformer may have been supplied by the antenna or cable installer. The other situation occurs when you have a television that was designed for 75-ohm coaxial cable instead of twin-lead. Here you will need the opposite forms of converter. Both types are available at moderate cost from television repair shops, or, from wholesalers whose main customers are TV repair shops. Be sure to set the switch box to "Computer" when using the computer, and "TV" when watching television.

2. The dual cord used to interconnect with the cassette tape recorder is color-coded. In the version which I received, one cord has a black miniature phone plug on each end, while the other has a red miniature phone plug on each end. It is important that you decide which will be used for MIC and which for EAR, and then maintain that protocol forever more. In may case, the red cord is used for the EAR, so one end is connected to the "EAR" input on the T/S-1000,

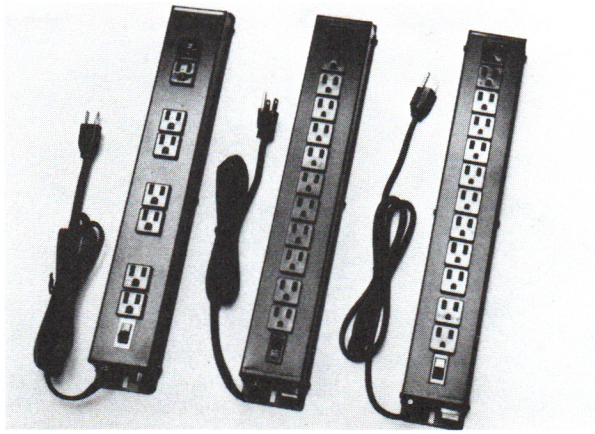


FIGURE 3-6. AC outlet strips.

while the other is connected to the "EAR" (some recorders use "EARPHONE") output on the cassette recorder; the black cord is connected between the "MIC" output on the T/S-1000 to the "MIC" (some use "MICROPHONE") input on the cassette player.

3. The power supply is contained within the bulbous assembly that also contains the 115V AC power line plug. This assembly is plugged into the wall outlet, and then into the 3.5 mm (VDC jack on the side of the T/S–1000. I believe it is good practice to plug in the computer end first, and then insert the supply into the wall socket. This is because there is almost inevitably a small spark as you insert the plug into the computer . . . and could theoretically lead to damage of the power supply, even though I have never seen it on this product. The power supply is not switched, so the T/S–1000 is turned on as soon as both ends of the power cord are connected. I prefer to plug the power supply AC plug into a switched outlet box so that it can be turned on and off at will without exercising those plugs. Simple two to four outlet switched strips can be purchased at hardware stores and radio wholesale shops; examples are shown in Fig. 3-6.

The 16K memory module will eventually be bought by all serious T/S-1000 users (unless they buy one of the larger modules by another vendor). This module (and all other accessories) plug into the rear interface connector on the T/S-1000 computer (see Fig. 3-7). IT IS VERY IMPORTANT that you not plug or unplug these accessories while power is applied to the computer. Always disconnect the power line (or turn off the switch) before attaching or removing any interface connector accessory!

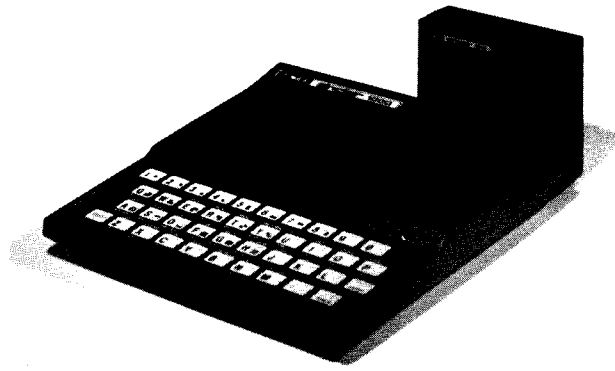


FIGURE 3-7. T/S-1000 with memory module attached.

If you have followed instructions properly, and power is applied to your computer, and the TV is properly adjusted to either ch-2 or ch-3 as you selected, then a small reverse-video "K" will appear in the lower left-hand corner of the screen. You are now ready to begin messing with the T/S-1000—or are you? FIRST, READ THE INSTRUCTION MANUAL THAT COMES WITH THE MACHINE! This little step is omitted by too many users, and can save you a lot of grief. Read at least through Chapter 5 of the T/S-1000 instruction manual, and then go back and work out some of the exercises which they recommend.

CHAPTER 4
USING THE TIMEX COMPUTERS

The three Timex computers all operate in a similar manner, and each comes with an instruction manual that instructs you quite well on the particulars. Therefore, we will not belabor this topic, except to make some general comments regarding the writing of programs.

So now you have the Timex set up and the "K" device appears in the left-hand corner of the television screen. Now what? I suppose the first stage is to learn a little something about BASIC programming (in Chapter 18 we will discuss how to enter machine language programs into the Timex. The BASIC language is the major method for programming the Timex, while the machine language is but a minor feature. BASIC permits English-like commands, but machine language, on the other hand, can be a lot more powerful where efficiency is a matter of some concern.

Operation of the Timex is covered in depth in the instruction manual that comes with the computer, so will not be repeated here in great detail. Our coverage here will be a matter of a few brief pointers to get you started.

When the "K" appears on the screen, the computer will accept keywords from the keyboard. Thus, when the "K" is on the screen, the data recognized by the computer in response to keystrokes will be the BASIC language keywords. If we press the "P" button, therefore, the computer will print on the screen not "P" but rather, "PRINT" (which is a BASIC command).

Following entry of a keyword, the computer reads keystrokes as letters, rather than keywords, until after an ENTER key is pressed.

WRITING A PROGRAM

Assuming that everything is connected correctly, and that you are familiar with the uses of the keywords, functions and other keyboard commands, it is time to begin to write programs. For that, incidentally, you are well-advised to consult a good book on BASIC computer programming, especially one which deals directly with the Timex. Also, read the instruction book for the Timex thoroughly. We can, however, give you a few pointers that will make the job a lot easier. This chapter, then, is grassroots philosphy.

Programming is a logical process, and does not in most cases require a genius. In fact, there was once a psychological profile performed on computer programmers, and it was compared with the profiles of other vocations. The surprise was that the computer programmer, except for those who operate in that netherworld out on the frontiers of human knowledge, should have an IQ just a little above average . . . certainly not genius class. The same personality profile which fits computer programmers also fits pastors, Catholic priests and police officers. I suspect the reason for this is that all four types of people will find themselves "enforcing" or living by rigid rules—the Holy Scripture, the Law of the Land or the BASIC computer language.

That little anecdote from my freshman Psyc-101 course of many years ago leads us into an admonishment concerning not just the Timex but *all* computers. The computer is a dumb neanderthalic brute that must be told exactly what to do, and behaves according to certain very strict, rigid laws that are, as far as the computer is concerned, immutable. Computer programs must be written in the precise syntax which the computer demands. For example, if the correct print command is PRINT, then the computer simply will not respond to spelling variants, such as PRNT, PRT or print (lower case of PRINT). Fortunately, on the Timex, many of these syntax problems are taken care of for us by the use of single-key keyword commands. There is still plenty of room for syntax errors, however, and that is where a thorough reading of the instruction manual comes in.

The reader who is used to dealing with reasonable humans, who seem to have an almost infinite capability to understand regardless of syntax errors, will be frustrated by the pig-headedness of the computer. For example, suppose the programmer wishes to print on the television screen the words: "HIYA PETE!" The print protocol demands that we give the command a line number, a BASIC command PRINT, and the text message inside quotation marks. There are two ways to do this job:

```
10  PRINT 'HIYA PETE!'
```

or,

```
10  PRINT "HIYA PETE!"
```

The first of these uses single quotation marks, while the second uses double quotation marks. Only the second version will print on the screen; the first will result in a syntax error signal ("S") in the lower left-hand corner of the screen.

The Timex version of BASIC will permit line numbers from 1 to 9999. You may at first be tempted to use sequential line numbers, because it looks so darn reasonable. But, when you find out just how easy it is for errors (or new requirements previously unthought of) to creep into the program, you will see the wisdom of skipping line numbers. The computer doesn't care that they are not in integer sequence, it will always execute the next higher instruction. Most programmers will start at 10 for the first line number, and then number the following in steps of ten:

```
10  PRINT "HIYA PETE"
20  PRINT "WHO'S PETE?"
30  PRINT "YOUR BOSS"
```

After writing line 30, we find that a critical statement has been left out. Since there is space between the lines, we can insert a line 25:

```
25  PRINT "WHO WANTS TO KNOW?"
```

Seriously, though, the use of spacing between the lines is a lifesaver in many cases. You will be surprised at how often programmers have to come back and patch a segment of a program because of some glitch that showed up later. Let's show an example of a real error; consider the program below.

```
 10  PRINT "I LOVE YOU BONNIE"
 20  LET X = X-1
 30  IF X = 0 THEN GOTO 300
 40  GOTO 10
300  STOP
```

The original intent of this Valentine's Day program is to print on the screen the intimate expression "I LOVE YOU BONNIE" a number of times indicated by the initial value of X. But if you try to execute this program, then you will find that X has no initial value, and an error message will appear on the screen of the television display. But, since we skipped line numbers

when writing this program, we can patch-fix it by adding the following code:

```
5   LET X = 10
```

The program will now read:

```
5   LET X = 10
10  PRINT "I LOVE YOU BONNIE"
20  LET X = X - 1
30  IF X = 0 THEN GOTO 300
40  GOTO 10
300 STOP
```

In this program we have initialized X at 10. The program will first print the endearment message intended, and will then decrement X by 1. The command LET X= X−1 simply means that *after the execution of the command,* the value of X will be 1 less than it had been before—it is not a violation of algebraic rules! If X is not 0, then the program will "fall through" to line 40, which tells it to go back to line 10 and again print the endearment message. If, on the other hand, the value of X is 0, then the computer will follow the conditional jump instruction in line 30 and go to line 300, which ends the program.

The screen will read:

```
I LOVE YOU BONNIE
I LOVE YOU BONNIE
I LOVE YOU BONNIE
I LOVE YOU BONNIE
I LOVE YOU BONNIE
I LOVE YOU BONNIE
I LOVE YOU BONNIE
I LOVE YOU BONNIE
I LOVE YOU BONNIE
I LOVE YOU BONNIE
```

(Would any readers care to guess the author's wife's name?)

The keyboard on the T/S-1000 is a membrane type which provides very little tactile or aural feedback to the user. Because of this problem, you will have to continually monitor the screen in order to make sure that the keystroke "took," and did not "double clutch" (print twice). This problem does not exist on the T/S-1500 and 2068.

PART II
INTERFACING HARDWARE

CHAPTER 5
Z80 ARCHITECTURE

The Z80 is an integrated circuit microprocessor designed and manufactured by Zilog, Inc. (10460 Bubb Road, Cupertino, CA 95014), and second-sourced by Mostek, Inc. (1215 West Crosby Rd., Carrolton, TX 75006) and others. The Z80 is similar to, but advanced over, the Intel 8080 microprocessor, In fact, a persistent industry story is that talent at Intel who designed the 8080 were the same people to design the Z80 device.

If you are familiar with the 8080 device, then making the switchover to Z80 will be very easy. The Z80 instruction set contains all of the 8080 instructions, plus a few more. It is usually claimed that the Z80 device has 158 different instructions, as opposed to only 78 for the 8080. Note that the means for numbering these instructions is a little less obvious, since in the last chapter we will introduce you to over 400 Z80 instructions. These "hidden" instructions are merely the expanded list (e.g., BIT, b, r can test any one of eight bits in any of seven different registers—making 56 instructions!).

In general, any program that will run on an 8080 system, with the exception of those dependent upon timing loops, will also run on a Z80 system. There are differences in the clock timing, so those programs that create, or are dependent upon, specific 8080 timing will not usually run properly on the Z80.

Besides the different instruction set sizes, there are other differences between the Z80 and the 8080. The programmer of the Z80 device can use

more internal registers and has more addressing modes than does the 8080 programmer.

In addition, there are several hardware differences. For one, the Z80 does away with the two-phase clock of the 8080. In the Z80, then, only a single-phase clock is used. The Z80 clock operates at 2.5 mHz, while the faster Z80A devices will accept clock speeds to 4 mHz or even 6.5 mHz. The Z80 also differs from the 8080 in that it will operate from a single +5-volt power supply. The 8080 device requires, in addition to the +5-volt supply, a −5-volt supply and a +12-volt supply.

The Z80 also provides an additional interrupt and the logic required to refresh dynamic memory.

The Z80 uses n-channel MOS technology, so must be handled with care in order to avoid damage from static electricity discharge.

Figure 5-1 shows the block diagram to the internal circuitry of the Z80 device. Note that the Z80 contains the following sections: arithmetic logic unit (ALU), CPU registers, and instruction register, plus sections to decode the instructions received and control the address placed on the address bus.

The Z80 uses an eight-bit data bus and a sixteen-bit address bus. The use of sixteen bits on the address bus means that the Z80 can address up to 65,536 different memory locations.

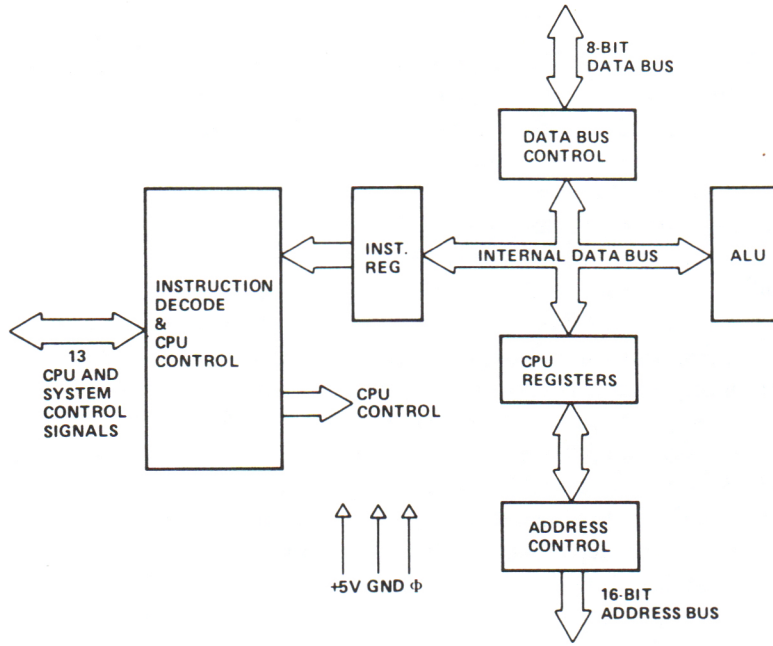


FIGURE 5-1. Z80 CPU block diagram.

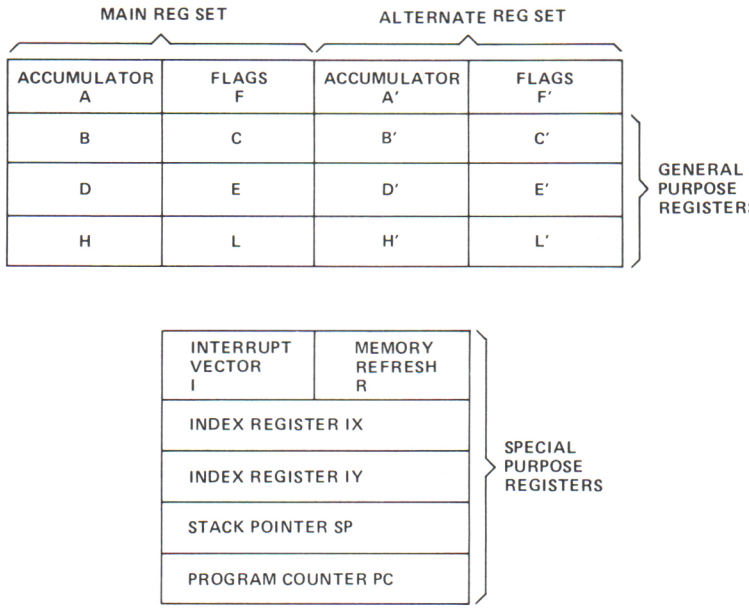


FIGURE 5-2. Register organization.

The internal registers of the Z80 represent 208 bits of read/write memory that can be accessed by the programmer. These bits are arranged in the form of eighteen 8-bit registers, and four 16-bit registers. Figure 5-2 shows the organization of the Z80 register set.

The main register set consists of an accumulator (register A) and a flag register (register F), plus six general-purpose registers (B, C, D, E, H, and L). An alternate set of registers is provided that duplicates these registers: accumulator (A′) and flag register (F′), plus the general-purpose registers B′, C′, D′, E′, H′, and L′. Only one set of these registers can be active at any one time. One cannot, for example, use the B and B′ registers without first using one of the instructions that interchanges the register sets.

The general-purpose registers can be paired to form three register pairs of 16 bits each: BC, DE, and HL. The alternate registers are also paired to alllow 16-bit register pairs BC′, DE′, and HL′.

The Z80 special-purpose registers include interrupt vector I and memory refresh R (both 8-bit registers), and four 16-bit registers: index register IX, index register IY, stack pointer SP, and program counter PC.

Interrupt Vector I. The I register is used to service interrupts originated by a peripheral device. The CPU will jump to a memory location containing the subroutine that services the interrupting device. The device will supply the

lower-order eight bits of the 16-bit address, while the I register will contain the high-order eight bits of the address.

Memory Refresh R. This register is used to refresh dynamic memory during the time when the CPU is decoding and executing the instruction fetched from memory. Seven bits of the R register are incremented after each instruction fetch, but the eighth bit remains as programmed through a LD R,A instruction. During refresh, a refresh signal becomes active, the contents of the R register are placed on the lower eight bits of the address bus, and the contents of the I register are placed on the upper eight bits of the address bus.

Index Registers IX and IY. These registers are used to point to external memory locations in indirect addressing instruction. The actual memory location addressed will be the sum of the contents of an index register and a displacement integer *d* (or, alternatively, some instructions use the two's complement of *d*). Both IX and IY index registers are independent of each other. Note that many microprocessor chips do not have index registers at all.

Stack Pointer (SP). The stack pointer is a two-byte register that is used to hold the 16-bit address of a last-in-first-out (LIFO) stack in external memory. The data to and from the memory stack are handled through the PUSH and POP instructions, respectively.

Program Counter (PC). The program counter in any computer holds the address of the instruction being fetched from memory. In the Z80, the program counter is a 16-bit register. The PC will be automatically incremented the correct number of digits after each instruction (e.g, one-byte instructions increment PC + 1, two-byte instructions PC + 2, etc.). When a JUMP operation occurs, the program counter will contain the address of the location to which the program jumped. When it is RETurned, the PC will contain the address of the next sequential instruction that would have been fetched if no jump had occurred.

Figure 5-3 shows how the program counter would work on a jump operation. Let us say that we have a program that starts at location 02 00 (hex), and finishes at location 02 06. But when it encounters the instruction at 02 02, it is an unconditional jump to location 06 12. Now, for the purposes of illustration, our subroutine at 06 12 is a RETurn instruction (useless in the real world, perhaps, but useful for illustration). It then jumps back to the next sequential location 02 05. Note that the next sequential location from 02 02 in this case is not 02 03, but 02 05. This is due to the fact that the jump instruction was a three-byte instruction. We had to give it the instruction (02 02), the low-order byte of the memory location to jump to (02 03), and the high-order byte of the memory location (02 04).

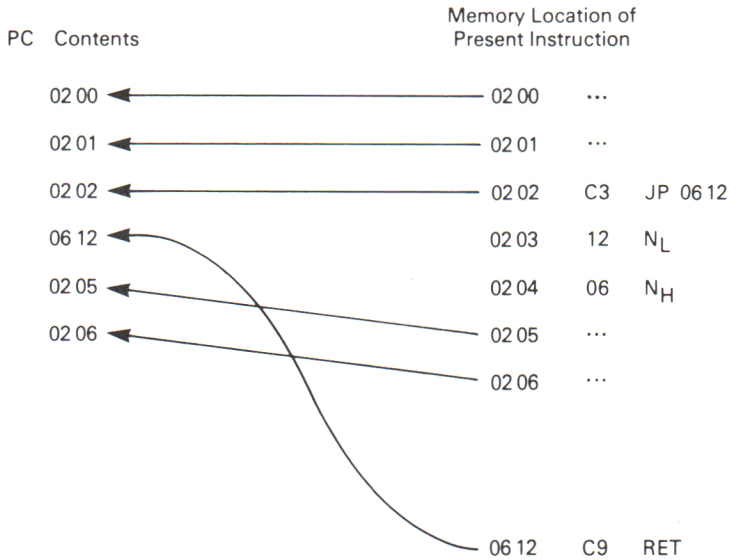


FIGURE 5-3.

ARITHMETIC LOGIC UNIT (ALU)

The heart of any computer or microprocessor, and the factor that distinguishes it from all other digital electronic circuits, is the arithmetic logic unit, or ALU. This circuit performs the data manipulation for the device. The functions possible in the Z80 uP are add, subtract, compare, logical AND, logical OR, logical exclusive-OR (XOR), left shift (logical), left shift (arithmetic), right shift (logical), right shift (arithmetic), increment, decrement, set a bit (i.e., make it 1), reset a bit (make it Ø), and test a bit to see whether it is 1 or Ø.

FLAG REGISTERS (F AND F')

The Z80 provides two status registers; F and F'. Only one is active at any one time, depending upon whether the programmer has selected the main register bank or the alternative register bank. These registers are each eight bits long and each bit is used to denote a different status condition. As a result, these bits of the F and F' register are also called *condition bits.*

The flags in the F and F' register are SET or RESET after certain arithmetic or other operations upon data. The program can then tell something about the result of the operation. The allocations are as follows:

BIT (F/F')	DESIGNATION	MEANING
Ø	C	Carry flag. Indicates a carry from the high-order bit of the accumulator (B7).
1	N	Subtraction flag used in BCD subtract operations.
2	P/V	Parity/overflow
3	X	Undetermined
4	H	BCD *half-carry* flag (bit 4 in BCD operations)
5	X	Undetermined
6	Z	Zero flag is SET if the result of an operation is zero.
7	S	Sign flag is SET if the sign of a result after an operation is negative, RESET if it is zero or positive.

Z80 PIN-OUTS

The Z80 device is constructed in a standard 40-pin DIP integrated circuit package. Since the Z80 uses NMOS technology, one is cautioned to become familiar with the rules for handling such devices before trying to handle the Z80 device. Those rules are actually very simple, so failing to follow them will net you what you deserve—a zapped IC.

Figure 5-4 shows the Z80 pin-outs and package configuration. The definitions of the pin-outs are given below, and the electrical (AC) specifications of the Z80 are given in Fig. 5-5.

Pin Definitions for the Z80

AØ-A15 Address bus (16 bits). Permits addressing up to 64K (i.e., 65,536 bytes) of memory, plus 256 different I/O ports. The address bus is active when HIGH, and has tri-state outputs. The entire 16 bits are used to address memory, while only the low-order byte (AØ–A7) is used to address I/O ports.

DØ-D7 Eight-bit data bus terminals. The data bus is, like the address bus, active high and uses tri-state outputs.

$\overline{M1}$ Machine cycle 1. When this terminal is LOW, the CPU is in the op-code fetch portion of the instruction/execution cycle.

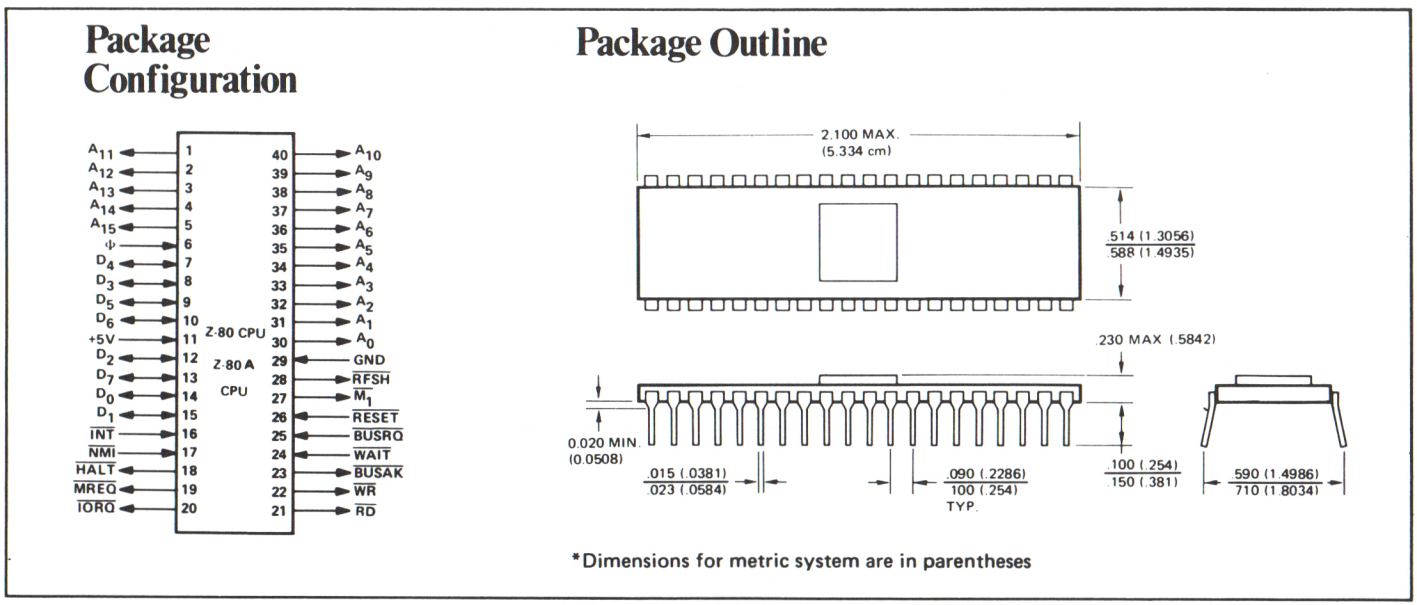


FIGURE 5-4. Z80 package and pinouts.

A.C. Characteristics Z80-CPU

T_A = 0°C to 70°C, Vcc = +5V ± 5%, Unless Otherwise Noted.

Signal	Symbol	Parameter	Min	Max	Unit	Test Condition
Φ	t_c	Clock Period	.4	[12]	μsec	
	t_w (ΦH)	Clock Pulse Width, Clock High	180	[E]	nsec	
	t_w (ΦL)	Clock Pulse Width, Clock Low	180	2000	nsec	
	$t_{r,f}$	Clock Rise and Fall Time		30	nsec	
A_{0-15}	$t_{D(AD)}$	Address Output Delay		145	nsec	C_L = 50pF
	$t_{F(AD)}$	Delay to Float		110	nsec	
	t_{acm}	Address Stable Prior to $\overline{MREQ}$ (Memory Cycle)	[1]		nsec	
	t_{aci}	Address Stable Prior to $\overline{IORQ}$, $\overline{RD}$ or $\overline{WR}$ (I/O Cycle)	[2]		nsec	
	t_{ca}	Address Stable from $\overline{RD}$, $\overline{WR}$, $\overline{IORQ}$ or $\overline{MREQ}$	[3]		nsec	
	t_{caf}	Address Stable From $\overline{RD}$ or $\overline{WR}$ During Float	[4]		nsec	
D_{0-7}	$t_{D(D)}$	Data Output Delay		230	nsec	C_L = 50pF
	$t_{F(D)}$	Delay to Float During Write Cycle		90	nsec	
	$t_{S\Phi(D)}$	Data Setup Time to Rising Edge of Clock During M1 Cycle	50		nsec	
	$t_{S\overline{\Phi}(D)}$	Data Setup Time to Falling Edge of Clock During M2 to M5	60		nsec	
	t_{dcm}	Data Stable Prior to $\overline{WR}$ (Memory Cycle)	[5]		nsec	
	t_{dci}	Data Stable Prior to $\overline{WR}$ (I/O Cycle)	[6]		nsec	
	t_{cdf}	Data Stable From $\overline{WR}$	[7]			
	t_H	Any Hold Time for Setup Time	0		nsec	
$\overline{MREQ}$	$t_{DL\overline{\Phi}(MR)}$	$\overline{MREQ}$ Delay From Falling Edge of Clock, $\overline{MREQ}$ Low		100	nsec	C_L = 50pF
	$t_{DH\Phi(MR)}$	$\overline{MREQ}$ Delay From Rising Edge of Clock, $\overline{MREQ}$ High		100	nsec	
	$t_{DH\overline{\Phi}(MR)}$	$\overline{MREQ}$ Delay From Falling Edge of Clock, $\overline{MREQ}$ High		100	nsec	
	$t_{w(\overline{MRL})}$	Pulse Width, $\overline{MREQ}$ Low	[8]		nsec	
	$t_{w(\overline{MRH})}$	Pulse Width, $\overline{MREQ}$ High	[9]		nsec	
$\overline{IORQ}$	$t_{DL\Phi(IR)}$	$\overline{IORQ}$ Delay From Rising Edge of Clock, $\overline{IORQ}$ Low		90	nsec	C_L = 50pF
	$t_{DL\overline{\Phi}(IR)}$	$\overline{IORQ}$ Delay From Falling Edge of Clock, $\overline{IORQ}$ Low		110	nsec	
	$t_{DH\Phi(IR)}$	$\overline{IORQ}$ Delay From Rising Edge of Clock, $\overline{IORQ}$ High		100	nsec	
	$t_{DH\overline{\Phi}(IR)}$	$\overline{IORQ}$ Delay From Falling Edge of Clock, $\overline{IORQ}$ High		110	nsec	
$\overline{RD}$	$t_{DL\Phi(RD)}$	$\overline{RD}$ Delay From Rising Edge of Clock, $\overline{RD}$ Low		100	nsec	C_L = 50pF
	$t_{DL\overline{\Phi}(RD)}$	$\overline{RD}$ Delay From Falling Edge of Clock, $\overline{RD}$ Low		130	nsec	
	$t_{DH\Phi(RD)}$	$\overline{RD}$ Delay From Rising Edge of Clock, $\overline{RD}$ High		100	nsec	
	$t_{DH\overline{\Phi}(RD)}$	$\overline{RD}$ Delay From Falling Edge of Clock, $\overline{RD}$ High		110	nsec	

[12] $t_c = t_{w(\Phi H)} + t_{w(\Phi L)} + t_r + t_f$

[1] $t_{acm} = t_{w(\Phi H)} + t_f - 75$

[2] $t_{aci} = t_c - 80$

[3] $t_{ca} = t_{w(\Phi L)} + t_r - 40$

[4] $t_{caf} = t_{w(\Phi L)} + t_r - 60$

[5] $t_{dcm} = t_c - 210$

[6] $t_{dci} = t_{w(\Phi L)} + t_r - 210$

[7] $t_{cdf} = t_{w(\Phi L)} + t_r - 80$

[8] $t_{w(MRL)} = t_c - 40$

[9] $t_{w(MRH)} = t_{w(\Phi H)} + t_f - 30$

$\overline{WR}$	$t_{DL\Phi}$ (WR)	$\overline{WR}$ Delay From Rising Edge of Clock, $\overline{WR}$ Low		80	nsec	C_L = 50pF
	$t_{DL\overline{\Phi}}$ (WR)	$\overline{WR}$ Delay From Falling Edge of Clock, $\overline{WR}$ Low		90	nsec	
	$t_{DH\overline{\Phi}}$ (WR)	$\overline{WR}$ Delay From Falling Edge of Clock, $\overline{WR}$ High		100	nsec	
	t_w ($\overline{WR}$L)	Pulse Width, $\overline{WR}$ Low	[10]		nsec	
$\overline{M1}$	t_{DL} (M1)	$\overline{M1}$ Delay From Rising Edge of Clock, $\overline{M1}$ Low		130	nsec	C_L = 50pF
	t_{DH} (M1)	$\overline{M1}$ Delay From Rising Edge of Clock, $\overline{M1}$ High		130	nsec	
$\overline{RFSH}$	t_{DL} (RF)	$\overline{RFSH}$ Delay From Rising Edge of Clock, $\overline{RFSH}$ Low		180	nsec	C_L = 50pF
	t_{DH} (RF)	$\overline{RFSH}$ Delay From Rising Edge of Clock, $\overline{RFSH}$ High		150	nsec	
$\overline{WAIT}$	t_s (WT)	$\overline{WAIT}$ Setup Time to Falling Edge of Clock	70		nsec	
$\overline{HALT}$	t_D (HT)	$\overline{HALT}$ Delay Time From Falling Edge of Clock		300	nsec	C_L = 50pF
$\overline{INT}$	t_s (IT)	$\overline{INT}$ Setup Time to Rising Edge of Clock	80		nsec	
$\overline{NMI}$	t_w ($\overline{NM}$L)	Pulse Width, $\overline{NMI}$ Low	80		nsec	
$\overline{BUSRQ}$	t_s (BQ)	$\overline{BUSRQ}$ Setup Time to Rising Edge of Clock	80		nsec	
$\overline{BUSAK}$	t_{DL} (BA)	$\overline{BUSAK}$ Delay From Rising Edge of Clock, $\overline{BUSAK}$ Low		120	nsec	C_L = 50pF
	t_{DH} (BA)	$\overline{BUSAK}$ Delay From Falling Edge of Clock, $\overline{BUSAK}$ High		110	nsec	
$\overline{RESET}$	t_s (RS)	$\overline{RESET}$ Setup Time to Rising Edge of Clock	90		nsec	
	t_F (C)	Delay to Float ($\overline{MREQ}$, $\overline{IORQ}$, $\overline{RD}$ and $\overline{WR}$)		100	nsec	
	t_{mr}	$\overline{M1}$ Stable Prior to $\overline{IORQ}$ (Interrupt Ack.)	[11]		nsec	

[10] $t_{w(\overline{WR}L)} = t_c - 40$

[11] $t_{mr} = 2t_c + t_{w(\Phi H)} + t_f - 80$

NOTES

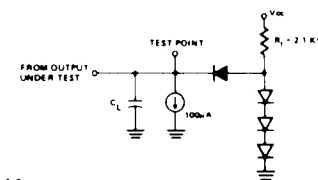
A Data should be enabled onto the CPU data bus when $\overline{RD}$ is active. During interrupt acknowledge data should be enabled when $\overline{M1}$ and $\overline{IORQ}$ are both active

B All control signals are internally synchronized, so they may be totally asynchronous with respect to the clock

C The $\overline{RESET}$ signal must be active for a minimum of 3 clock cycles.

D Output Delay vs. Loaded Capacitance
TA = 70°C Vcc = +5V ±5%
Add 10nsec delay for each 50pf increase in load up to a maximum of 200pf for the data bus & 100pf for address & control lines

E Although static by design, testing guarantees $t_{w(\Phi H)}$ of 200 μsec maximum



Load circuit for Output

FIGURE 5-5. Z80 electrical (AC) specifications.

$\overline{\text{MREQ}}$ Memory request signal. Is active low, and is an active low output. When this terminal goes low, the address on the address bus is valid for a memory operation (read or write).

$\overline{\text{IORQ}}$ Input/output request. This active low, tri-state output indicates that an I/O operation is to take place. The low-order byte of the address bus (AØ-A7) contains the address (0-255) of the selected port. The contents of the accumulator may be placed on the high-order byte of the address bus during this period. The $\overline{\text{IORQ}}$ is also generated to acknowledge an interrupt request, and tells the interrupting device to place the interrupt vector word on the data bus (i.e., low-order byte of the address of the interrupt service program.)

$\overline{\text{RD}}$ This is an active, low, tri-state output that indicates when a read operation from memory, or an I/O device, to the CPU is taking place.

$\overline{\text{WR}}$ Tri-state, active, low output that indicates when a write operation from the CPU to a memory location, or I/O device, is taking place. Tells the memory or I/O device that the data on the data bus are currently valid.

$\overline{\text{RFSH}}$ Refresh signal. This is an active low output that indicates that the lower seven bits of the address bus contain a refresh address for the dynamic memory.

$\overline{\text{HALT}}$ Active low output that indicates that a halt instruction is being executed. The CPU executes NOPs while in the halt state, and is awaiting the receipt of an interrupt signal.

$\overline{\text{WAIT}}$ Active low input that indicates that the addressed memory, or I/O device, is not yet ready to transfer data to the data bus.

$\overline{\text{INT}}$ Active low input that tells the CPU that an external device has requested an interrupt. The CPU will honor the request at the end of the current instruction cycle, if the interrupt flip-flop (software controlled) is SET.

$\overline{\text{NMI}}$ Active low input for nonmaskable interrupt operation. This line will cause the CPU to honor the interrupt at the end of the current instruction cycle, regardless of the state of the interrupt flip-flop. Forces automatic restart at location 00 66 (hex).

$\overline{\text{RESET}}$ Active low input that enables the interrupt flip-flop, clears the program counter (i.e., loads PC with 00 00) and clears

I and R registers. This terminal can serve as a hardware jump-to-00-00 control.

$\overline{\text{BUSRQ}}$ Active low input that requests that the CPU address bus, data bus, and the control signals go to the high impedance (tri-state) state so that some other device can obtain control of these buses. The $\overline{\text{BUSRQ}}$ has a higher priority than $\overline{\text{NMI}}$, and is always honored at the end of the present instruction.

$\overline{\text{BUSAK}}$ Active low output that is used with the bus request signal, and tells the requesting device that the CPU buses are now in the high impedance state. When $\overline{\text{BUSAK}}$ drops low, then the requesting device may take control of the buses.

Φ Clock signal input. Wants to see TTL level at 2 mHz or 4 mHz (Z80A) maximum.

GND DC and signal ground terminal.

+5 Power supply terminal, to which is applied +5 volts DC from a regulated power supply.

Z80 TIMING AND INTERFACE CONTROL SIGNALS

If you are planning to use a ready-built computer (like the T/S-1000) containing a Z80 microprocessor chip, then it is not likely that you will need to know much about the chip-level interface and timing signals of the chip unless you plan to do some interfacing. Languages like BASIC, and even some assemblers, will not require that you know much at all about these signals. But if you are doing machine level programming, using most assemblers, or are trying to interface some other instrument to the Z80 directly, or to the bus of a Z80 computer, it is then necessary for you to know and understand the timing structure.

Earlier we discussed the definitions of the Z80 pins. Among the pins discussed were the interface and timing signals. For emphasis, let us reiterate these signals here, but grouping them according to use.

Data/Address Buses

There are two buses in the Z80: a 16-bit address bus and an 8-bit data bus. The address bus pinouts are labeled AØ–A15, while the data bus terminals are designated BØ–B7. In both cases, the Ø bit is the least significant bit, while the highest numbered bit (7 on the data bus, 15 on the address bus) is the most significant bit.

Both address and data buses are designed to be tri-state outputs. This means that there are the HIGH and LOW states for logical 1 and Ø, respectively, plus a third high impedance state which can be used to effectively disconnect the Z80 CPU chip from the external bus lines. In some cases, a bus request signal (discussed below) will cause the data and address buses to go into the tri-state condition, so that an external device can control the buses.

Also, both address and data buses are active when HIGH.

The data bus is used to pass data to, and from, the CPU chip. Unless one knows the status of the control signals, and the word applied to the address bus, however, one does not know what is taking place on the data bus.

The address bus does several things. In the memory address mode, for example, the 16-bit address bus will be capable of designating 2^{16}, or 65,536, different memory locations. This size, incidentally, is usually called "64K," even though 65,000+ locations are addressable. This is due to the fact that a computer "K" is 1024, not 1000 (sigh).

The address bus is also used in the control of input/output operations. When an I/O command is being executed, the lower byte of the address bus holds the address of the I/O port designated in the instruction. The upper byte contains the accumulator data, repeated on the data bus.

The lower byte of the address bus is also used in the memory refresh operation. During the period of the machine cycle in which the refresh operation is to take place, as indicated by a LOW condition on the $\overline{\text{RFSH}}$ output terminal, the lower seven bits (AØ–A6) of the address bus contain the refresh address.

Input/Output (I/O) Operations

The Z80 design philosophy is a little different from the philosophy of its direct ancestor, the 8080 device. This is especially noticeable in the I/O operations. In the Z80, there is an *input/output request* ($\overline{\text{IORQ}}$) signal available. This is a tri-state, active low output that is used to tell external devices and memory that an input or output operation is taking place.

The $\overline{\text{IORQ}}$ signal will go LOW when (a) an input or output operation is taking place, and (b) when an interrupt is being acknowledged. In the latter case, an $\overline{\text{M1}}$ signal is also generated during interrupts. This combination of signals is used to tell the interrupting device to place the address vector pointing to the interrupt service subroutine. These two types of operation can be distinguished from each other because interrupt acknowledgments always occur during the M1 period (see below), and I/O operations never occur during the M1 period.

It is not possible to use just one signal for I/O control, because there are three possible states: no I/O operation, input, and output. In the first case, the $\overline{\text{IORQ}}$ line would be HIGH, but it will be LOW for both of the other possible

conditions. In the Z80 device, the input and output states are distinguished by the condition of the $\overline{WR}$ and $\overline{RD}$ control signal. These are also used in memory operations, and are the *write* ($\overline{WR}$) and *read* ($\overline{RD}$) signals. If the I/O operation is an input (i.e, read), then the $\overline{RD}$ line goes LOW along with $\overline{IORQ}$. But if the I/O operation is an output, then the $\overline{WR}$ control signal goes LOW along with $\overline{IORQ}$.

Memory Control Signals

Control of memory operations requires the same $\overline{WR}$ and $\overline{RD}$ signals as used in the I/O operations. But instead of the $\overline{IORQ}$ signals, a *memory request* ($\overline{MREQ}$) is used. This signal is an active low, tri-state output that is used to indicate that the address bus contains a valid memory location address. Whether the CPU is reading from memory or writing to memory is indicated by the coincidence of the $\overline{MREQ}$ and $\overline{RD}$ (memory read), or $\overline{MREQ}$ and $\overline{WR}$. Address decoders in memory, then, must take note of these signals in order to determine whether a read or write operation is taking place.

There is also a *refresh* ($\overline{RFSH}$) signal used to control dynamic memories. Unlike static memory devices, dynamic memory often requires a refresh operation, or the data stored will be lost. $\overline{RFSH}$ is an active low, tri-state output, and is active once during each instruction fetch operation. When the $\overline{RFSH}$ and $\overline{MREQ}$ are both low, a memory refresh can take place. The contents of the R register are loaded onto the lower seven bits of the address bus to address the memory to be refreshed. The R register is incremented after each operation, so all memory will eventually be refreshed.

CPU Control Signals

There are four basic CPU control signals: $\overline{M1}$, $\overline{RESET}$, $\overline{WAIT}$, and $\overline{HALT}$.

The $\overline{M1}$ signal is used to indicate that an M1 instruction fetch period is in effect. The M1 machine cycle occurs when an instruction is being fetched from memory. If the instruction being fetched is a two-byte instruction, then an $\overline{M1}$ signal is generated as each op-code is being fetched.

The $\overline{M1}$ signal is also generated during interrupt acknowledgments, in conjunction with an $\overline{IORQ}$ signal. This combination allows the interrupting device to place the address vector of the memory location containing the interrupt service subroutine.

The $\overline{RESET}$ signal is an active low input. When this terminal is brought low, the CPU does the following: Disables the interrupt flip-flop; sets the I register to ØØ (hex); sets the R register to ØØ (hex), and sets interrupt mode Ø. In effect, the $\overline{RESET}$ is a *hardware* jump to ØØ ØØ instruction.

The $\overline{WAIT}$ terminal is an active low input that can be used to tell the

CPU that an addressed I/O device is not ready to transfer data. The CPU keeps entering wait states until this signal goes high again. This signal is needed because many types of I/O device are not as fast as the CPU.

The $\overline{\text{HALT}}$ signal is an active low output that indicates that a halt instruction is being executed. The CPU will execute no-ops (NOP) until an interrupt is received.

Interrupt Signals

The principal interrupt signals are the $\overline{\text{INT}}$ and $\overline{\text{NMI}}$. The regular *interrupt request* signal is the $\overline{\text{INT}}$. It is an active low input. The interrupt request signal is generated by the interrupting I/O device. The interrupt request will be honored at the end of the present instruction cycle. There are three modes of response by the CPU: mode Ø, mode 1, and mode 2.

The nonmaskable interrupt ($\overline{\text{NMI}}$) signal is used to allow interrupts that *must* be serviced at the end of the current instruction cycle.

$\overline{\text{BUSRQ}}$ and $\overline{\text{BUSAK}}$

These signals are used to allow access to the memory by external devices, without the use of the CPU. The $\overline{\text{BUSRQ}}$ is an active low input. When the $\overline{\text{BUSRQ}}$ line goes low, the CPU outputs (address and data buses) go tri-state at the end of the current instruction cycle.

The $\overline{\text{BUSAK}}$ is an active low output that tells the external device that the CPU is in the high impedance tri-state condition. When this signal goes low, the external device knows that it now has control of the data and address buses.

Basic CPU Timing

All instructions in any programmable digital computer are merely a series of certain basic operations. In discussing the timing of the CPU, we must determine how these operations occur. The clock produces periods called T-periods (see Fig. 5-6). There are also three different "M cycles" for each instruction cycle. M cycles are machine cycles, while the T cycles are clock cycles.

Machine cycle M1 is the instruction fetch period, and may be four to six T cycles long. During this period, the CPU is fetching the next instruction from the memory. The other machine cycles (M2 and M3), are used for memory read and memory write, respectively. M2 and M3 are used for memory and I/O operations.

In the paragraphs to follow, we will discuss the *op-code instruction*

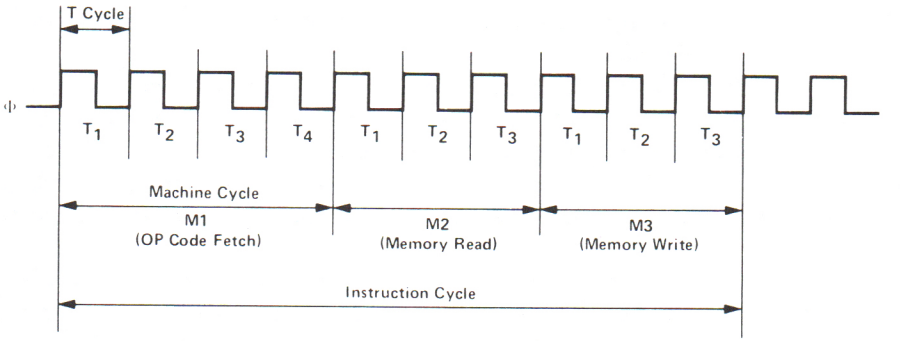


FIGURE 5-6. Timing diagram for instruction cycle.

fetch, memory data read/write, I/O read/write, bus request/acknowledge, interrupt request, nonmaskable interrupt request, and *exit from HALT instruction cycles.*

Op-code Instruction Fetch. Figure 5-7 shows the CPU timing during the M1 op-code instruction fetch cycle of the Z80 CPU. The program counter (PC) contains the address of the next instruction. The contents of the PC are placed on the address bus (AØ–A15) during the first half of the M1 cycle.

Since we are trying to fetch, i.e., *read*, an instruction from some location in memory, the $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$ signals are also placed low. This tells the

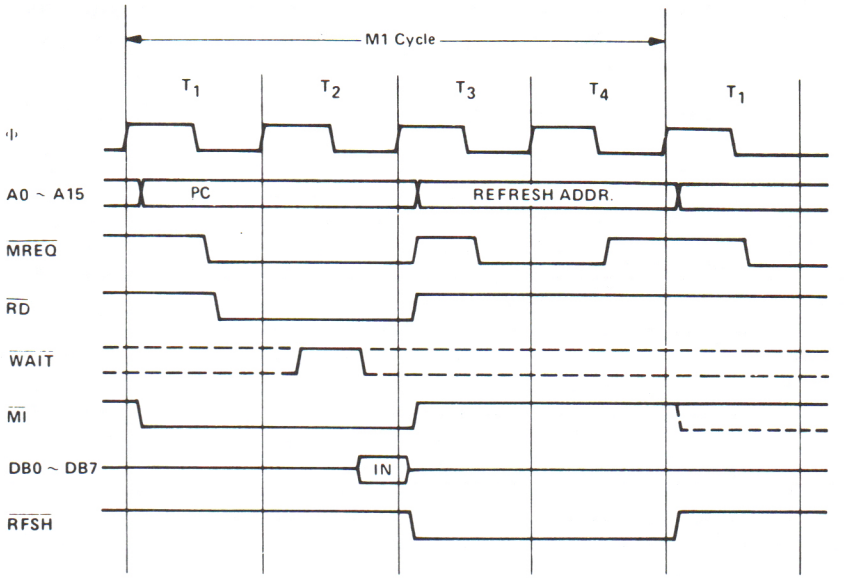


FIGURE 5-7. M1 cycle timing diagram.

memory that a read operation is taking place from a location whose address is found on the address bus.

The $\overline{\text{WAIT}}$ line is sampled during this period. If the memory device is slow, it may generate a wait signal to slow down the operation. If a wait signal is found during each sample (i.e., once during each T cycle), then the CPU will enter another wait state. When the device is ready to transfer data, the wait signal disappears, and the data bus contains the data from that memory location.

During the last half of the M1 cycle (i.e., T3/T4), the refresh address is placed on the lower seven bits of the address bus, and a $\overline{\text{RFSH}}$ is generated. This will alow the refreshing of dynamic solid-state memories.

During the portion of the M1 cycle that the program counter contents are on the address bus, the $\overline{\text{M1}}$ signal is low.

The M1 machine cycle will lengthen for as long as there is a wait signal present. Using the $\overline{\text{WAIT}}$ line permits us to synchronize the CPU and an external device.

Memory Data Read/Write. The M2 and M3 machine cycles are used to read to, or write from, memory locations. Figure 5-8 shows the CPU timing during these operations. The principal signals used in this type of operation are the $\overline{\text{MREQ}}$, $\overline{\text{WR}}$, and $\overline{\text{RD}}$.

If a memory read operation is needed, then an address is placed on the address bus (AØ–A15) during the M2 machine cycle. During this period, the $\overline{\text{MREQ}}$ (memory request) and $\overline{\text{RD}}$ lines coincidentally go low. The $\overline{\text{MREQ}}$ signal does not become active until the data on the address bus are stable.

Memory write operations cause data from the CPU to be written into

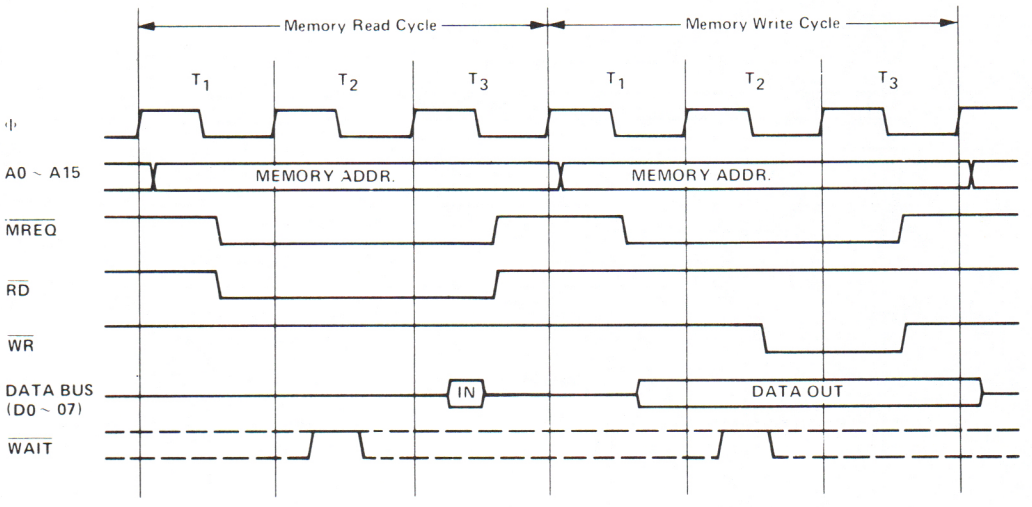


FIGURE 5-8. Memory read/write cycle timing.

specified locations in memory. This occurs during the M3 machine cycle. In this operation, the MREQ and WR signals become active. The MREQ signal, however, does not become active until the data on the data bus are stable (i.e., valid), so that semiconductor memory can be accommodated. Again, the address of the specified location is applied to the address bus (AØ–A15).

As in the instruction fetch cycle, a wait state can be created. If the WAIT signal is low, then the CPU continues to enter wait states until the signal becomes inactive. The WAIT signal can be used to synchronize the CPU to memory sources.

I/O Read/Write. Figure 5-9 shows the CPU timing during input and output cycles. During each of these types of operation, the IORQ request line becomes active (i.e., goes low). If the operation is a read cycle, then the RD signal will also go low. But if the operation is a write cycle (i.e., an output), then the WR signal goes low coincidentally with IORQ.

In both input and output cycles, the address of the designated port is placed on the lower byte of the address bus (AØ–A7). Since this is an eight-bit address, we can specify up to 256 different addresses from ØØØ–255 (decimal).

During an input (i.e., I/O read) operation, the IORQ and RD signals are low during T2 and T3, and data from the input port are passed along the data bus.

During an output (i.e., I/O write) operation, the IORQ and WR signals are low during T2 and T3. Data from the accumulator are passed over the data bus to the output port whose address is contained on the lower byte of the address bus. But note that the IORQ signal does not become active

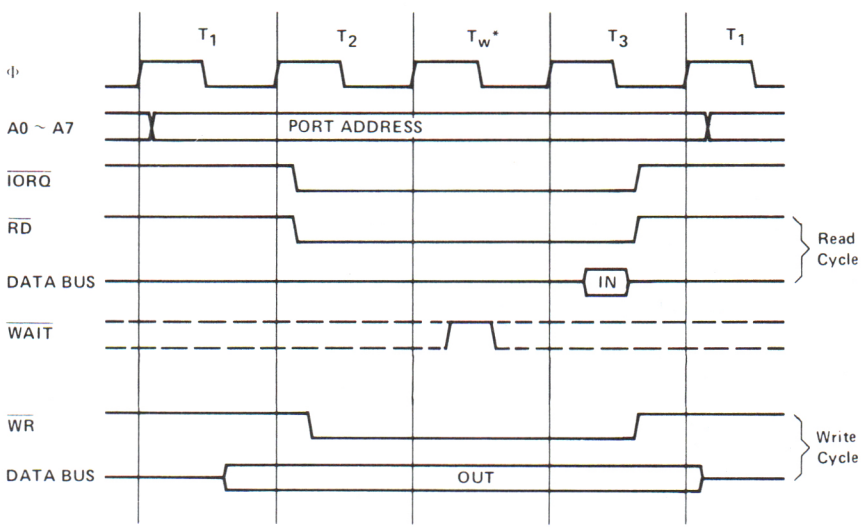


FIGURE 5-9. I/O timing diagram.

immediately, allowing the data on the data bus to stabilize before the operation is consummated.

Bus Request/Acknowledge. The bus request signal ($\overline{BUSRQ}$) is used to allow external devices to gain control of the CPU control lines, the address bus and the data bus. This allows direct access to memory for the external device.

The CPU samples the $\overline{BUSRQ}$ input during the last T cycle of any given M cycle. If the bus request is active, the CPU will complete the current instruction, and then service the request. Following the last T cycle of the last M cycle, the CPU will go into a high impedance state. The address bus lines, the data bus lines, and the control lines ($\overline{MREQ}$, $\overline{RD}$, $\overline{WR}$, $\overline{IORQ}$, $\overline{RFSH}$) are placed in the high impedance condition, effectively disconnecting them from the external circuits. This will allow the external device to gain control of the lines, to directly input data to memory locations without going throught the CPU. When the CPU lines are in the high impedance state, the CPU generates a $\overline{BUSAK}$ (bus acknowledge) signal that tells the requesting device that the buses are available to its use. The timing for this type of operation is shown in Fig. 5-10.

When the external device is finished with the memory, it will deactivate (i.e., make high) the $\overline{BUSRQ}$ signal, telling the CPU that it can have control again.

Interrupt Request. The ability to service interrupts allows the CPU to use certain types of external device more efficiently. The CPU can do other chores while the slower external device is working, or it may perform other chores while awaiting rarely occurring situations to develop. The $\overline{INT}$ signal

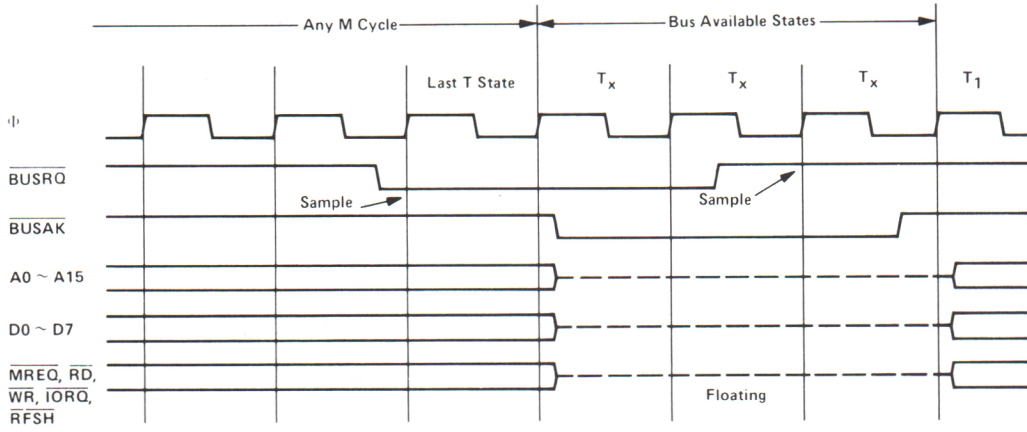


FIGURE 5-10. Bus request timing diagram.

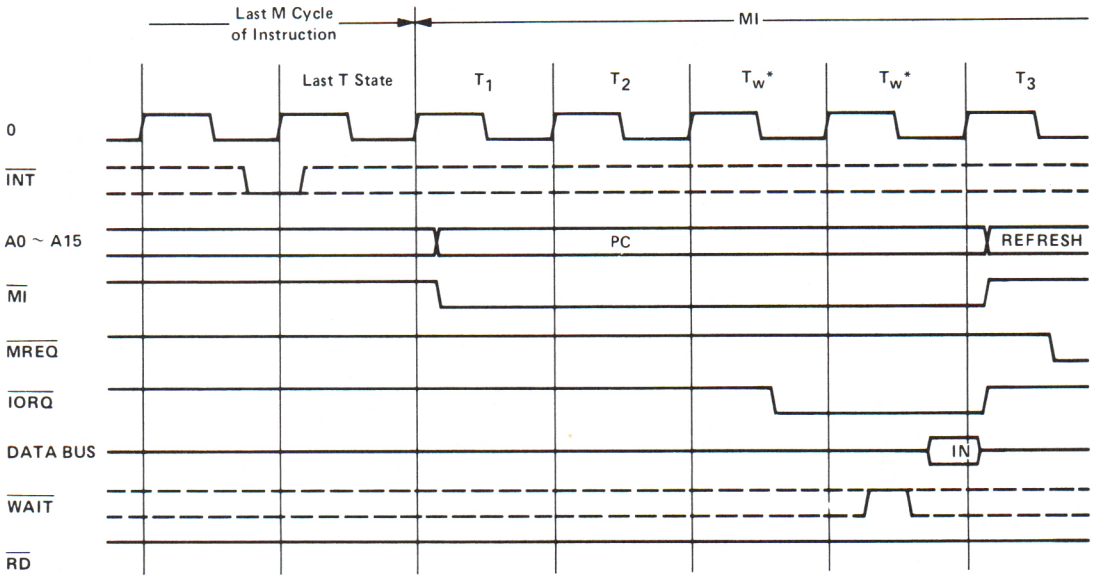


FIGURE 5-11. Interrupt (INT) timing diagram.

is the interrupt request. This line is sampled by the CPU on the rising edge of the last T state of each M cycle. See Fig. 5-11.

These interrupts can be masked in software because the CPU will not accept the request unless an internal CPU flip-flop is set. This interrupt flip-flop is controlled by software commands. Interrupts are also ignored if the bus request ($\overline{\text{BUSRQ}}$) line is active (i.e., low).

If the CPU accepts the interrupt request, then a special M1 state is generated, so the $\overline{\text{M1}}$ line goes low. The address bus receives the contents of the program counter (PC), so that the CPU can return to the original program after the interrupt is serviced. The address of the next instruction, to be executed following termination of the interrupt, is stored on an external memory stack.

Once the PC contents are stored, the $\overline{\text{IORQ}}$ line goes low, telling the interrupting device that it can place an address vector on the data bus, which tells the CPU where the program that services the interrupt is located.

As in the previous conditions, the $\overline{\text{WAIT}}$ signal can be used to lengthen the timing by causing the CPU to enter wait states. If the $\overline{\text{WAIT}}$ line is active when sampled, the CPU enters the wait state. If the signal is inactive, then no wait state is generated, and the CPU continues.

Nonmaskable Interrupts. Certain types of interrupt situations cannot wait for the software being executed by the CPU to set an internal flip-flop. Such interrupts might be an alarm condition in the process or factory, or in a

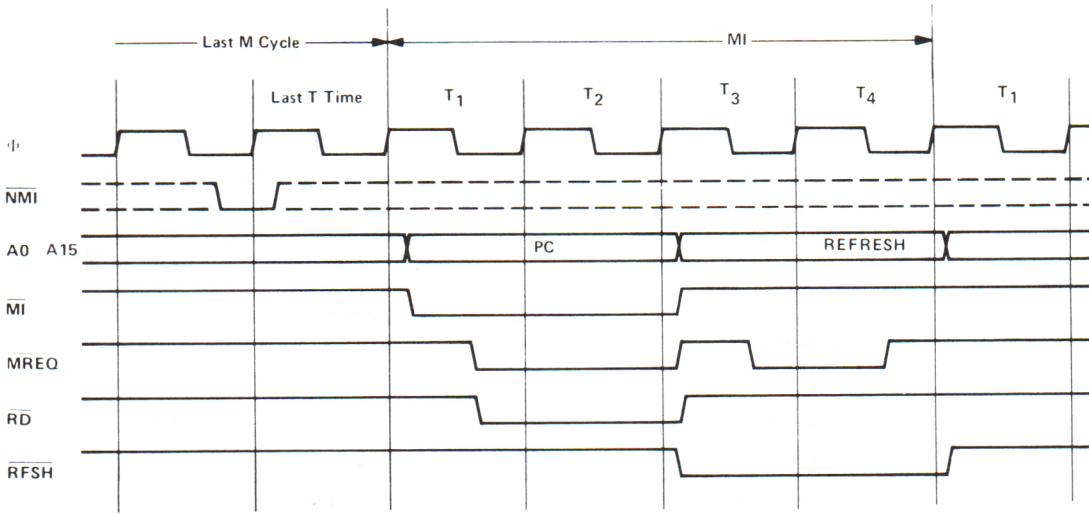


FIGURE 5-12. Interrupt (NMI) timing diagram.

medical computer. These situations require a nonmaskable interrupt. Figure 5-12 shows the CPU timing for the nonmaskable interrupt in the Z80.

This type of interrupt cycle is very similar to the regular interrupt, except that it is not dependent upon the software-controlled interrupt flip-flop. This type of interrupt will be serviced as soon as the present instruction cycle is completed. The contents of the program counter are stored in an external memory stack, and the CPU jumps automatically to location ØØ 66 (hex) to find the interrupt service program.

Exit from a HALT. If a software HALT instruction is encountered, the CPU will sit there executing no-ops (NOP) until one of two situations occurs: (a) a nonmaskable interupt is received, or (b) a maskable interrupt is received, and the internal interrupt flip-flop is SET.

If the Interrupt lines ($\overline{INT}$ or $\overline{NMI}$) are active when sampled during the T4 portion of the M1 cycle, then the HALT condition is terminated following T4. The $\overline{HALT}$ line then goes inactive (high). The CPU timing for the exit from halt operation is shown in Fig. 5-13.

Z80 FLAGS

The F and F′ registers in the Z80 CPU chip are used as condition flags. Six of the eight bits in each register are SET (i.e., made equal to 1) or RESET (i.e., made equal to 0), depending upon the conditions resulting from various CPU

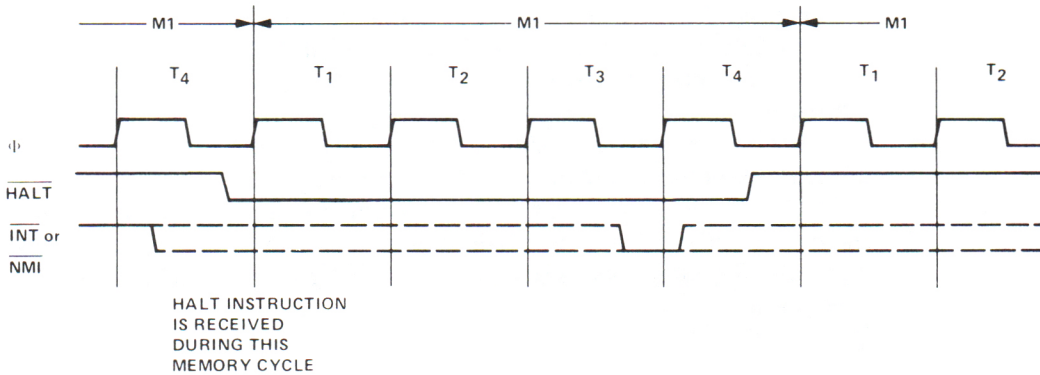


FIGURE 5-13. Halt timing diagram.

operations. The flag register contents are available to the programmer. The bits of the flag registers are assigned as follows:

BIT NO.	DEFINITION
0	C
1	N
2	P/V
3	(undefined)
4	H
5	(undefined)
6	Z
7	S

The C (carry) flag tells us if there was a carry bit from the highest-order bit in the accumulator. The flag will be set if there is a *carry* from an addition operation, or *borrow* during a subtraction operation, or in certain cases, during the execution of the shift and rotate instructions.

The Z (zero) flag will be set if the operation performed results in a zero's (i.e., 00000000_2) being loaded into the accumulator. If any number other than zero results from the operation, then the Z flag is reset (i.e., made 0).

The S (sign) flag stores the state of bit 7 in the accumulator (i.e., 1 or 0). This results from the fact that bit 7 represents the sign of the number. If the number is negative, then bit 7 is a 1, but if it is zero or positive, then bit 7 is 0.

The P/V (parity/overflow) flag has two purposes. When logical operations (AND, OR, XOR) are performed, the P/V flag indicates the parity of the contents of the accumulator (odd or even). When two's complement operations are being performed, the P/V flag indicates whether

or not an overflow occurs. An overflow condition indicates that the answer in the accumulator is in error, because it has exceeded the maximum permissible range of numbers (−128 to +127). The P/V will be set if the overflow occurs, but there is no carry (C flag).

In the case of logical operations, the P/V flag will be set for even parity of the result in the accumulator, and reset for odd parity.

The H (half-carry) flag is a BCD borrow or carry from the least significant half-byte (½-byte = nybble?) of the accumulator contents. Similarly, the subtract flag (N) is used for correcting (decimal adjusting) BCD subtract operations.

CHAPTER 6
Z80 I/O

Input/output (I/O) devices include such apparatus as keyboards, teletypewriters, printers, digital displays, analog-to-digital converters (ADC), digital-to-analog converters (DAC), or mass memory storage devices such as disc drives or cassette drives (e.g., Phi-Decks). The Z80 uses the lower-order byte of the address bus to address I/O ports. We may, therefore address 2^8, or 256 different ports, which are given numerical assignments in the range ØØØ to 255. If port ØØ6 is addressed, for example, the address will be Ø6 (hex), or ØØØØØ11Ø in binary.

Z80 I/O CONTROL SIGNALS

In chapter 5 we defined the various control signals generated and/or recognized by the Z80 microprocessor chip on the T/S-1000, these signals are available at the interface connector on the rear of the case (see Chapter 11). Here we will review those that are relevant to the I/O functions. The $\overline{\text{IORQ}}$ (i.e., input/output request) signal is an active LOW, tri-state output. When this terminal (Z80 pin 20) goes LOW, the outside world knows that the lower byte of the address bus contains a valid input or output port address. There is also one additional use of the $\overline{\text{IORQ}}$ signal. It will go LOW during the M1 machine cycle when an interrupt is being acknowledged, so that external I/O

devices will know to place an interrupt response vector (i.e., the lower-order byte of the address at which the interrupt service subroutine may be located) on the data bus.

The $\overline{IORQ}$ signal occurs for both input and output operations. This could cause some confusion, so the Z80 designers, in their infinite wisdom, elected to also provide a read ($\overline{RD}$) and a write ($\overline{WR}$) signal. These signals occur for both memory and I/O operations, and allow the external circuitry or devices to determine whether we are reading to or from the CPU. When the $\overline{RD}$ and $\overline{IORQ}$ control lines are simultaneously LOW, the external world knows that the command being executed is an *I/O read* (i.e., an input) operation. Similarly, if the $\overline{IORQ}$ and the $\overline{WR}$ are simultaneously LOW, then the external world knows that an *I/O write* (i.e., an output) operation is to take place.

Z80 I/O INSTRUCTIONS

The Z80 I/O instructions are a powerful group that allows both block and direct commands. The accumulator (i.e., A register) can be used in either input or output immediate commands. Both types are two-byte instructions in which the accumulator either inputs from, or outputs to, the port whose address is given by the byte following the op-code.The other registers, however, use register indirect instructions, in which the address of the input or output port is given by the contents of the C register. *IN B, C,* for example, loads register B with the binary word at an input port whose address is given by the contents of the C register.

The block transfer instructions allow us to transfer whole blocks of sequentially stored data using but one instruction. These instructions are INI, INIR, INDR, OUTI, OTIR, OUTD, and OTDR. Many programmers feel that these are among the most powerful of the Z80 instructions.

I/O PORT ADDRESS DECODERS

Address decoders for I/O ports are very similar to those used for memory ports, with the exception that only eight-bit addresses are anticipated (as opposed to 65,536 memory addresses!). This will simplify the design process, but keep in mind that the same circuits as were used for the memory address can be pressed into service for decoding I/O addresses.

The problem is this: The $\overline{IORQ}$, in conjunction with $\overline{RD}$ and $\overline{WR}$, tells the I/O devices that an I/O operation is to take place. But just which I/O port is being designated? The address passed over the lower eight bits (AØ–A7) of the address bus must be decoded. Each I/O device, or I/O port, will contain

its own address decoder circuit. This circuit will tell the device to respond (as appropriate) to the $\overline{IORQ}/\overline{WR}$ or $\overline{IORQ}/\overline{RD}$ conditions.

The first address decoder is shown in Fig. 6-1, and is one of the most popular circuits used. The 7430 IC is a TTL eight-input NAND gate. Its output will remain HIGH as long as any one of the eight inputs is LOW. The output terminal of the 7430 will go LOW only if all eight inputs are HIGH. The trick is to make the entire set of inputs HIGH when the correct address is present on the lower eight bits of the address bus. Of course, if the address is

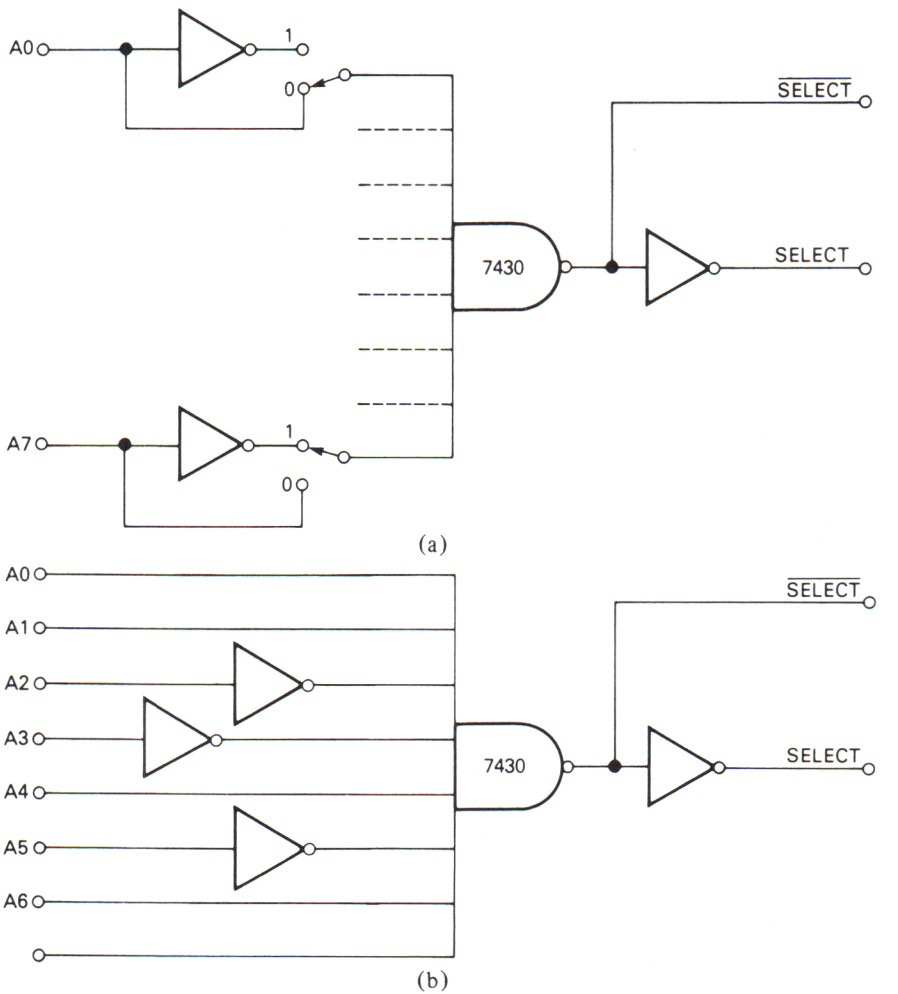


FIGURE 6-1. (a) Switches select whether "1" or "0" is required to generate a SELECT pulse. Only the LSB and MSB positions are shown in detail; (b) Decoder for address selected.

FF (hex) (11111111 in binary), then we have no problem. Connecting one each of the 7430 inputs to one of the lower-order bits of the address bus will automatically give us our decoder. But all other addresses will require one or more inverters between the address bus lines and the inputs of the 7430. If you want maximum flexibility, then one inverter may be dedicated to each 7430 input. But this is a terrible waste of inverters, because there is only one address in which all of the inverters are required: ØØØØØØØØ (binary)! All other addresses will use fewer than eight inverters. As a practical matter, most commercially available I/O printed circuit boards have but three or four inverters. The user is then asked to carefully select I/O port addresses such that no more than three or four zeroes occur. It is rare, indeed, that all 256 possible I/O ports would be required so this is not the sacrifice that it might appear. In the example of Fig. 6-1 we have shown inverters only on the AØ and A7 lines, with those for the other lines implied. In actual practice, most designers have the inverters wired in with small jumpers, rather than formal switches, so that they may be dedicated to *any* 7430 input as might be required.

By way of illustration, let us assign the address 11010011 (D3 in hex) to an I/O port. We see by inspection that all but three of the bits in this address are ones, so no inverters will be needed for them. Only the zero bits (i.e., A2, A3, and A5) will require inverters. The decoder for this address is shown in Fig. 6-1(b). Notice that the AØ, A1, A4, A6, and A7 lines are connected directly to 7430 inputs, while the A1, A2, and A5 address lines are passed through inverters before being applied to the 7430 inputs. When the address 11010011 appears on the bus, then, all of the 7430 inputs will see ones, and the 7430 can drop LOW. This creates a $\overline{\text{SELECT}}$ signal for use by the I/O circuitry. An optional inverter will turn this signal upside down, creating a SELECT signal, for those cases where a positive-going transition is needed.

Another form of address decoder is shown in Fig. 6-2. This circuit is based on the TTL-type 7485 comparator IC. This device will compare two

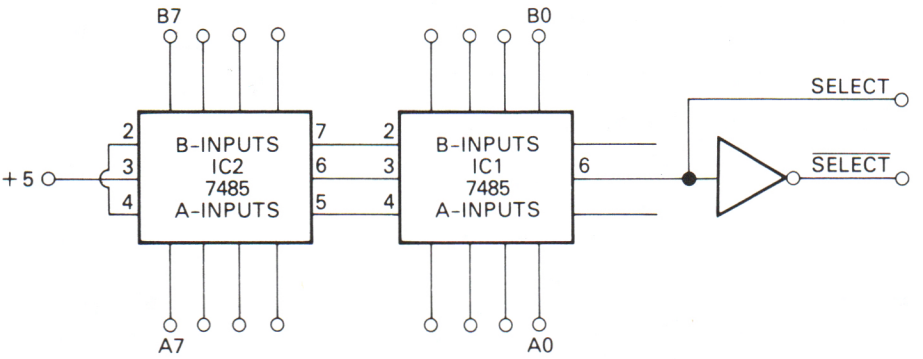


FIGURE 6-2. Using 7485 TTL comparators as an address decoder.

four-bit words ("A" and "B") and issue an output that indicates whether A is equal to, greater than, or less than B. Of these, we are interested in the A = B output (pin 6). The 7485 has cascading inputs that sense the status of a lower-order four bits. We need two 7485 devices connected in this cascade manner in order to use it to decode an eight-bit address bus.

We connect the bits of the address bus to the A-inputs of the 7485s. The B-inputs are used to program the device with the address of the port to be selected. In the previous case, we selected port D3 (hex), i.e., 11Ø1ØØ11 in binary and 211 in decimal, using a 7430. If we wanted to use the 7485's as shown in Fig. 6-2, we would program IC1 with the binary word ØØ11 (i.e., 3 hex), and IC2 with 11Ø1 (i.e., D hex). When this address appears on the address bus, then the A = B output of IC1 will go HIGH, forming a SELECT signal. An inverter is needed if a $\overline{\text{SELECT}}$ is desired instead.

We can also use any of the various "1-of-N" decoder ICs as address selectors. The 7442 is a 1-of-10 decoder, while the 74154 device is a 1-of-16 device. Each of these has a four-bit binary input to determine which output line (Ø through 9, or Ø through 15) will go LOW. Figure 6-3 shows the use of the 7442 1-of-10 decoder. Two 7442 devices are needed. A NOR gate is connected so that one input of the NOR gate is driven by one of the outputs of each 7442.

As an example, let us say that we want to use the circuit of Fig. 6-3 to decode address 115 (decimal), which is 73 in hex. The binary code for "7" is 0111, and the code for "3" is 0011. We want, then, to see binary code 01110011 on the AØ–A7 lines. We connect the four-bit inputs of IC1 to the lower-order four bytes of the address line (AØ–A3), and the high-order four-bits (A4–A7) to the four-bit inputs of IC2. One input of the NOR gate is then connected to the "3" output of IC1, and the other input is connected to the "7" output of IC2. When the correct address appears, both of these outputs will drop LOW, causing the output of the NOR gate to snap HIGH. This signal then becomes our SELECT signal. Again, an inverter is used to form a $\overline{\text{SELECT}}$ signal where required.

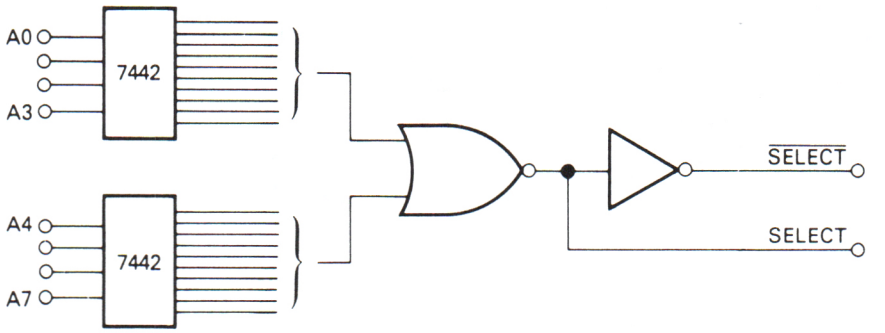


FIGURE 6-3. Using 7442 1-of-10 decoder as an address decoder.

GENERATING PORT/DEVICE SELECT SIGNALS

Computers based on microprocessors that provide separate I/O function generally require a means to *decode* the *port address* and sense whether an *input* or *output* operation is to take place. The Z80 microprocessor, for example, uses the following signals to define I/O operations: (a) an eight-bit address on the lower byte (AØ–A7) of the 16-bit address bus to specify the *port number,* (b) an active-LOW input/output request ($\overline{IORQ}$) line to indicate that an I/O operation is taking place, and (c) active-LOW *read* ($\overline{RD}$) and *write* ($\overline{WR}$) lines to indicate whether the request is an input or an output, respectively.

Figure 6-4 shows a method for generating an active-HIGH OUT or IN signal in Z80 based microcomputers. This circuit is based on a single 7402 IC (quad two-input NOR gate). Gates G3 and G4 are used to create the OUT and IN signals, and are controlled by the $\overline{IORQ}$ signal and the signals from *read* and *write* lines. Recall the rules governing NOR gate operation: a HIGH on any input produces a LOW output, and it requires all inputs be LOW for the output to be HIGH. When the $\overline{IORQ}$ line is inactive (i.e., HIGH), then the NOR gate outputs (G3 and G4) are forced LOW. If, however, $\overline{IORQ}$ becomes active (i.e., LOW), then the NOR gate outputs are unlatched and will be HIGH or LOW depending upon the state of the $\overline{RD}$ and $\overline{WR}$ lines.

Case No. 1: Input Operation. An input port operation is a read operation because data is transferred *into* the accumulator. When the Z80 device executes an input operation, therefore, the $\overline{IORQ}$ and $\overline{RD}$ go LOW. A LOW condition on $\overline{IORQ}$ unlocks gate G4 by bringing input-A LOW. The $\overline{RD}$ line, and the $\overline{SELECT}$ line from the address decoder, controls gate G2. When *both*

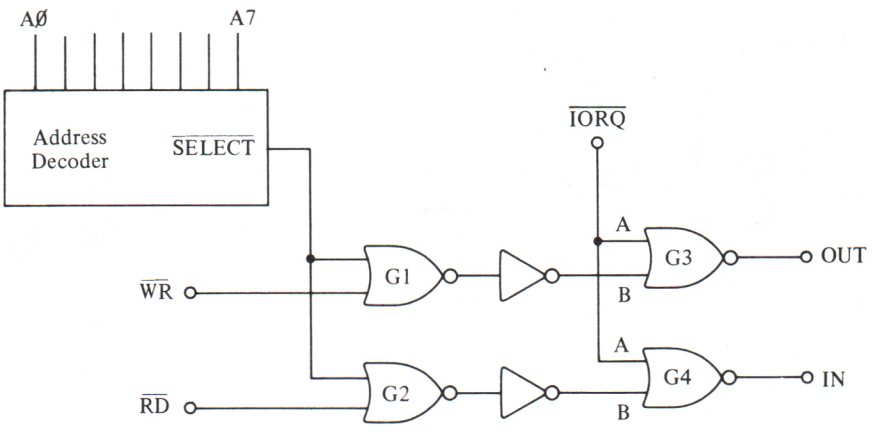
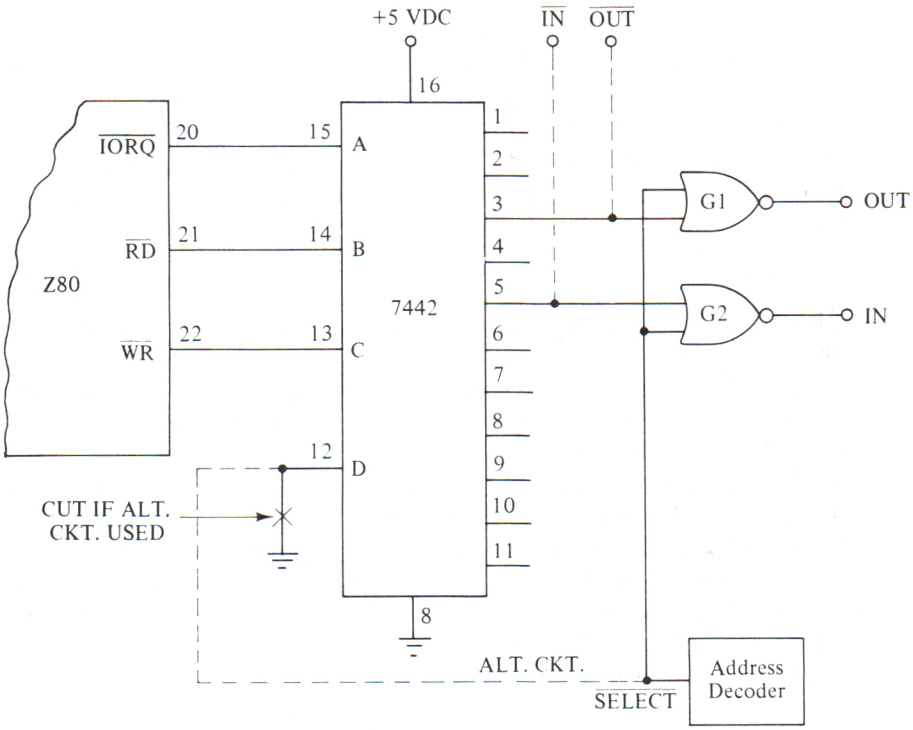


FIGURE 6-4. Simple IN and OUT pulse generator.

of these lines are LOW, then, input-B of G4 will also go LOW. We now find that both inputs of G4 are LOW, so the IN line (G4 output) goes HIGH.

Case No. 2: Output Operation. An output operation is a write function because data is transferred *from* the accumulator to the outside world. When a Z80 microprocessor executes an output operation, therefore, the $\overline{\text{IORQ}}$ and $\overline{\text{WR}}$ go LOW. A LOW condition on $\overline{\text{IORQ}}$ unlocks gate G3 by bringing input-A LOW. The $\overline{\text{WR}}$ line and the $\overline{\text{SELECT}}$ line from the address decoder controls gate G1. When both of these lines are LOW, then input-B of G3 will also go LOW. We now find both inputs of G3 LOW, so the OUT line goes HIGH.

A *select* circuit based on the 7442 TTL IC is shown in Figure 6-5. The 7442 is a TTL BCD-*to-1-of-10 decoder*. It will examine the four-bit binary coded decimal (BCD) input line and generates a unique active-LOW output



	Binary	Decimal
Input	0100	4
Output	0010	2

FIGURE 6-5. Using the 7442 as a device select pulse generator.

that depends on the decimal equivalent of the input word. For example, when the binary word 0100 (4_{10}) appears on the inputs, then the "4" output (pin no. 5) goes LOW.

Two methods are shown in Figure 6-5 for using the 7442 in this application. The main circuit is shown in solid lines, while the alternate circuit is shown in dotted lines.

The main circuit in Figure 6-5 works by considering the $\overline{IORQ}$, $\overline{RD}$, and $\overline{WR}$ lines from the Z80 as a three-bit binary word, and these lines are connected to the *A, B,* and *C* lines of the 7442, respectively. The *D* input line of the 7442 is wired permanently LOW. An input operation causes the $\overline{IORQ}$ and $\overline{RD}$ to be LOW, so the word applied to the 7442 is 0100, or decimal 4. This situation causes pin no. 5, which is the "4" output, to go LOW. Similarly, an output operation causes $\overline{IORQ}$ and $\overline{WR}$ to be LOW, so the word applied to the 7442 is 0010, or decimal 2. This condition causes pin no. 3 to go LOW during the output operation.

Like the previous circuit, Figure 6-5 is used with specific I/O ports, so it must recognize the signal from the address decoder. The $\overline{SELECT}$ line will go LOW when the CPU selects its address; at all other times this line is HIGH. The $\overline{SELECT}$ line is connected to both G1 and G2.

When pin no. 3 of the 7442 goes LOW, and the $\overline{SELECT}$ line is also LOW, then the output of gate G1 will go HIGH, providing an OUTPUT signal. This signal is used to tell the output port circuitry to accept data on the bus.

An INPUT signal is created by gate G2 under similar circumstances. The $\overline{SELECT}$ line will go LOW if the address for that port is indicated. Gate G2 knows that an input operation is demanded because pin no. 5 of the 7442 goes LOW in response to the binary code 0100 at its inputs.

In the alternate version of this circuit we can get rid of the two NOR gates by using the $\overline{SELECT}$ line from the address decoder to activate the D input of the 7442. This input wants to see a LOW condition before the $\overline{IN}$ and $\overline{OUT}$ lines are activated (again, pins 5 and 3 are used). If another I/O port is being called up, the A-B-C inputs will still see the correct code from the Z80 (010 and 100), but the D input sees a "1." The codes applied to the 7442 then are 1010 and 1100—so they will not activate any 7442 output line. When this port is selected, however, the 7442 input codes are either 0010 or 0100, so the appropriate output is activiated.

Figure 6-6 shows a circuit that uses just two NOR gates to create the IN and OUT signals. The gates selected for this circuit are three-input NOR gate sections of a CMOS 4000 IC. This circuit is simple and will work with most microcomputer systems. It will not work, however, in high speed (10 mHz and up) clock systems. These NOR gates require all three inputs LOW for the output to be HIGH. The only time this situation will occur is during input operations for IC1A and output operations for IC1B.

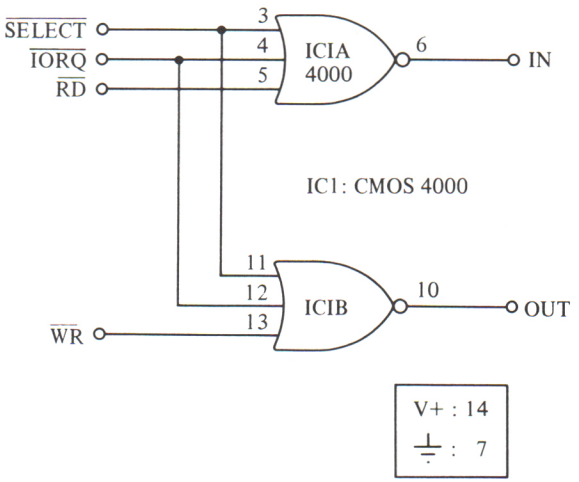


FIGURE 6-6. CMOS NOR gates used as a device select pulse generator.

Thus far, all of the circuits presented have been for specified I/O ports, i.e., an address decoder was included in the design. It is often desirable, however, to create *system I/O* signals that are active for any input or output operation. We then provide a simple address decoder at each port or on each printed circuit card within the microcomputer. This structure is popular in microcomputers that allow for future expansion through the addition of plug-in I/O cards. In a Z80 based system, we can create the system signals by adapting the circuit of Figure 6-6. We would either ground the 4000 device inputs used for the $\overline{\text{SELECT}}$ signal, or use two-input NOR gates such as the 7402 device.

PARALLEL OUTPUT PORT DESIGN

Although many microcomputers use special-purpose IC devices for I/O ports, it is often less costly to use ordinary IC devices from the regular digital logic families in these applications.

A Type-D flip-flop can be used to store one bit of data, so an array of eight Type-D flip-flops can be used to hold one byte of data. Figure 6-7(a) shows the symbol for the Type-D flip-flop. Data applied to the *D*-input will be transferred to the Q output only when the clock line is HIGH. If the clock line remains HIGH, then the data on the Q output will follow the data on the D input. If, however, the clock line is LOW, then the data on the Q output remains the same; i.e., the FF will hold the last valid D-input data present before the clock dropped LOW. The flip-flop will then *ignore* all further changes of the D-input data.

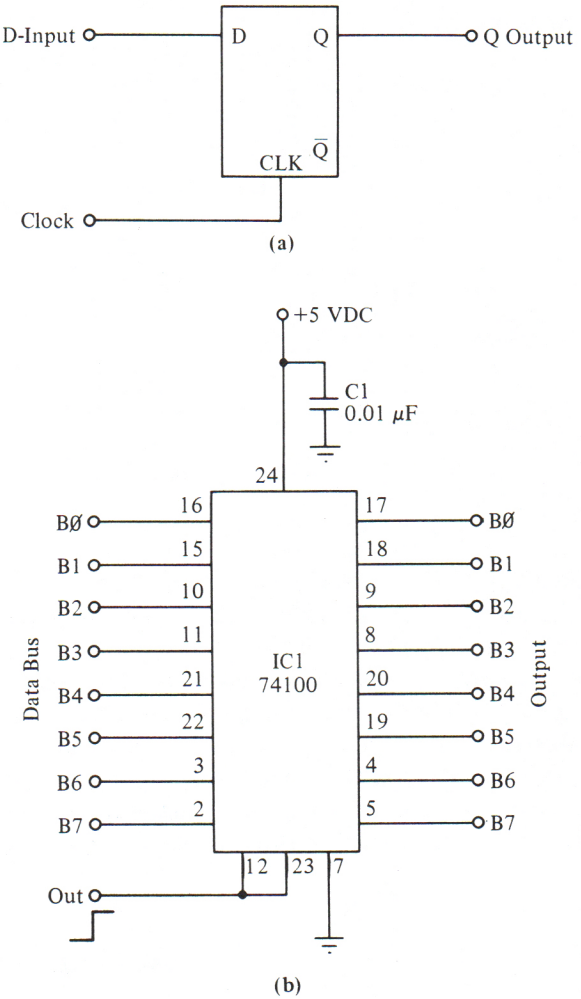


FIGURE 6-7. (a) Type-D flip-flop. (b) Latched output porting using 74100 dual quad latch.

There are a number of type-D flip-flops available in the major digital logic IC families. Certain devices called *data latches* contain four, or eight, Type-D flip-flops in arrays that share a common clock line. An example is the 74100 device shown as an output port register in Figure 6-7(b).

The 74100 is a dual four-bit data latch, and it contains a pair of Type-D flip-flop arrays—each containing four flip-flops. If the two clock lines (pins 12 and 23) are tied together, then the 74100 operates as an eight-bit register. The eight D-inputs are connected to the data bus, while the eight Q outputs

are used as the output bits. This type of circuit forms a latched output because the data remains fixed on the output lines after the execution of an output instruction is completed.

The common clock line from the 74100 is connected to the OUT signal from the port select circuit. The data on the data bus is transferred to the output side of the 74100 register whether the OUT signal line goes HIGH.

PARALLEL INPUT PORT DESIGN

Ordinary IC devices may also be used in the design of input ports. There is, however, a design constraint that limits our selection of devices to those which provide *tri-state* output lines.

The microcomputer data bus serves several functions, including transfers to and from many memory locations and an I/O port. If any one device causes a permanent HIGH or LOW condition on any line of the data bus, then there will be massive errors in the data. All devices connected to the data bus must, therefore, be tri-state devices.

A tri-state device is one that becomes effectively disconnected from the output terminal when it is not active. All digital ICs are at least two-state, i.e., the output terminal sees either a low impedance to ground (LOW condition), or, a low impedance to +5 volts DC (HIGH condition). The tri-state device has a third state in addition to the regular states in which the output terminal sees a high impedance to both ground and +5 volts DC anytime the *chip enable* (CE or $\overline{\text{CE}}$) terminal is inactive. There are only a few devices from the regular TTL lines that meet this criterion, an example of which is the 74125 device.

The 74125 is a four-bit TTL device that contains four noninverting buffers each of which has its own active-LOW chip enable ($\overline{\text{CE}}$) line. When $\overline{\text{CE}}$ is LOW, the buffer acts like any TTL buffer, so the output data directly follows the input data. However, when the $\overline{\text{CE}}$ line is HIGH, the buffer output line goes to a high impedance, and the device is effectively disconnected from the output line.

Figure 6-8 shows a pair of 74125 devices used to form an eight-bit input port. Since an input port should not provide latching, we cannot use the 74100 device. All four $\overline{\text{CE}}$ lines on both 74125 devices are tied together to form a master $\overline{\text{CE}}$ line that turns the port on and off in response to the $\overline{\text{IN}}$ signal.

The $\overline{\text{IN}}$ line will normally be held HIGH until an input operation takes place from that port. At that time, the $\overline{\text{IN}}$ line drops LOW, turning on the 74125s. This action connects the bits of the input ports to the data bus as long as $\overline{\text{IN}}$ is LOW, so input data appears briefly on the data bus.

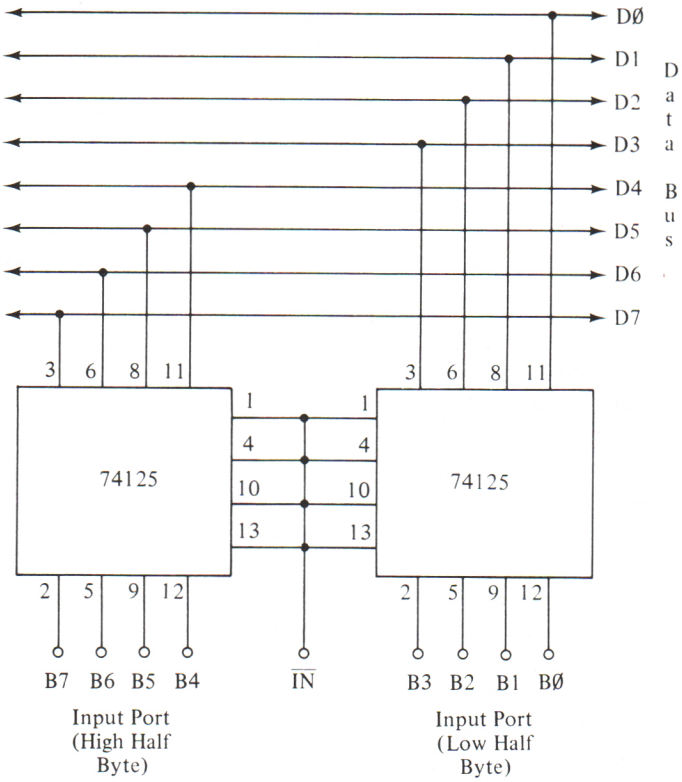


FIGURE 6-8. Input port using two 74125 chips.

BIDIRECTIONAL PARALLEL I/O PORT DESIGN

The job of designing and interfacing I/O ports is made easier by the use of certain special purpose IC I/O devices. Some of these are general enough to be used with a wide variety of microprocessor chips. The 8216/8226 and 8212 devices are examples. Produced by Intel for use with the 8080A microprocessor chip, they are frequently adapted for use with other CPU chips.

The 8216 and 8226 devices are almost identical to each other, except that the 8216 contains *noninverting* line drivers, while the 8226 contains *inverting* line drivers. Both are four-bit devices, so two are needed to drive an eight-bit data bus.

Figure 6-9 shows the logic diagram for the 8216 device. This same diagram is also used for the 8226, except that the line driver buffers are inverters. Lines DBØ through DB3 are connected to four bits of the data bus. The four input port lines are labelled DIØ through DI3, while four output port lines are designated DOØ through DO3.

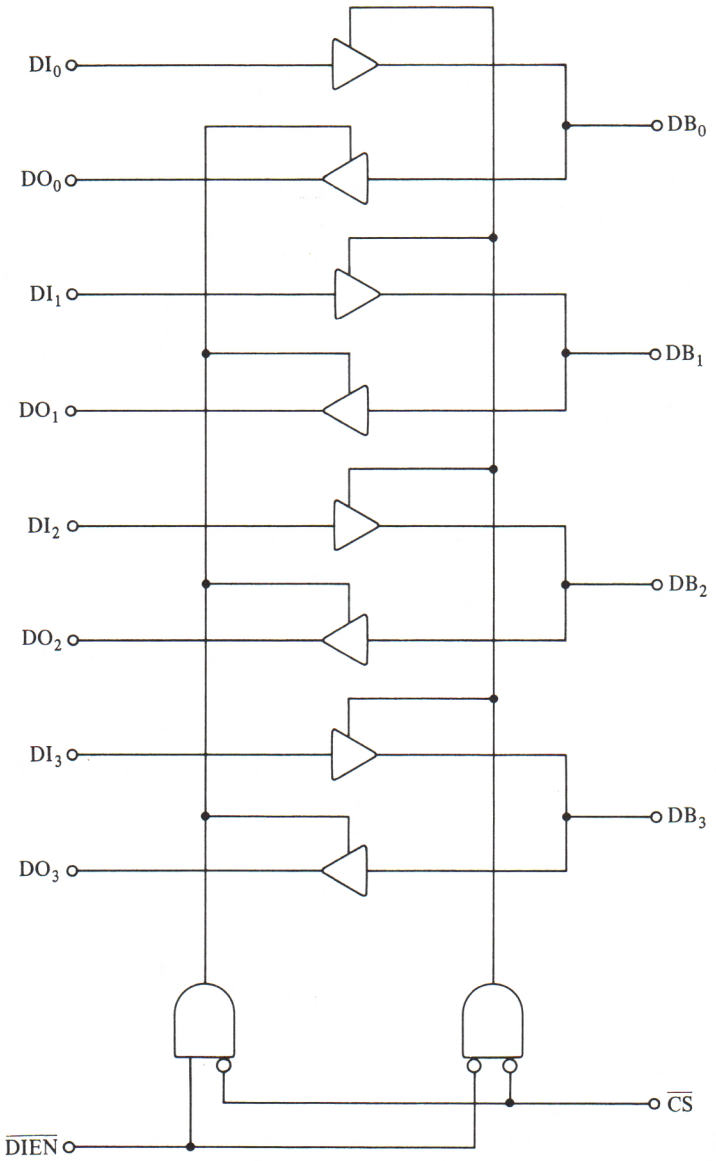


FIGURE 6-9. 8216 I/O port.

The $\overline{\text{DIEN}}$ line controls the data direction. If this line is LOW, then the input lines are connected to the data bus. Alternatively, if $\overline{\text{DIEN}}$ is HIGH, then the output port is connected to the data bus.

The lines connected to the data bus have tri-state outputs, so they can be made to float at high impedance when not needed. The chip select $\overline{\text{CS}}$ is an

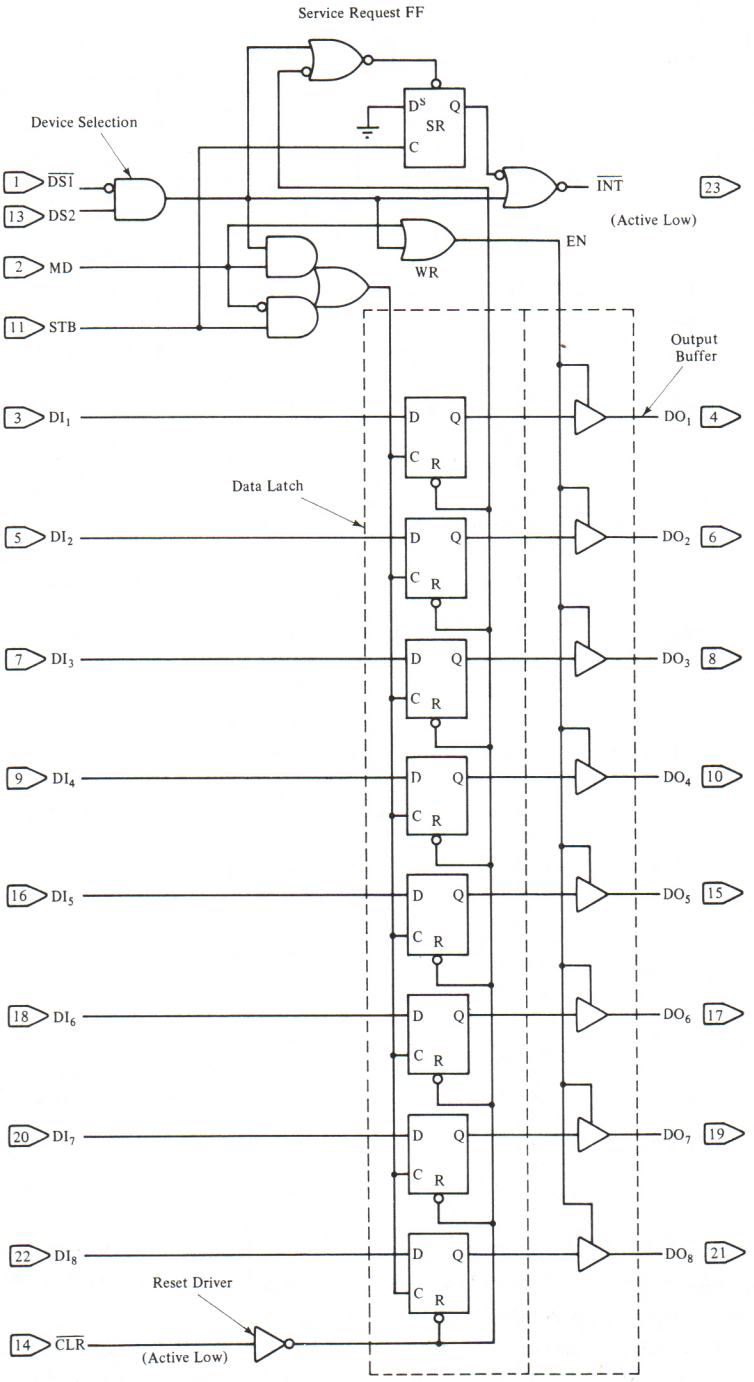


FIGURE 6-10(a). 8212 I/O port device.

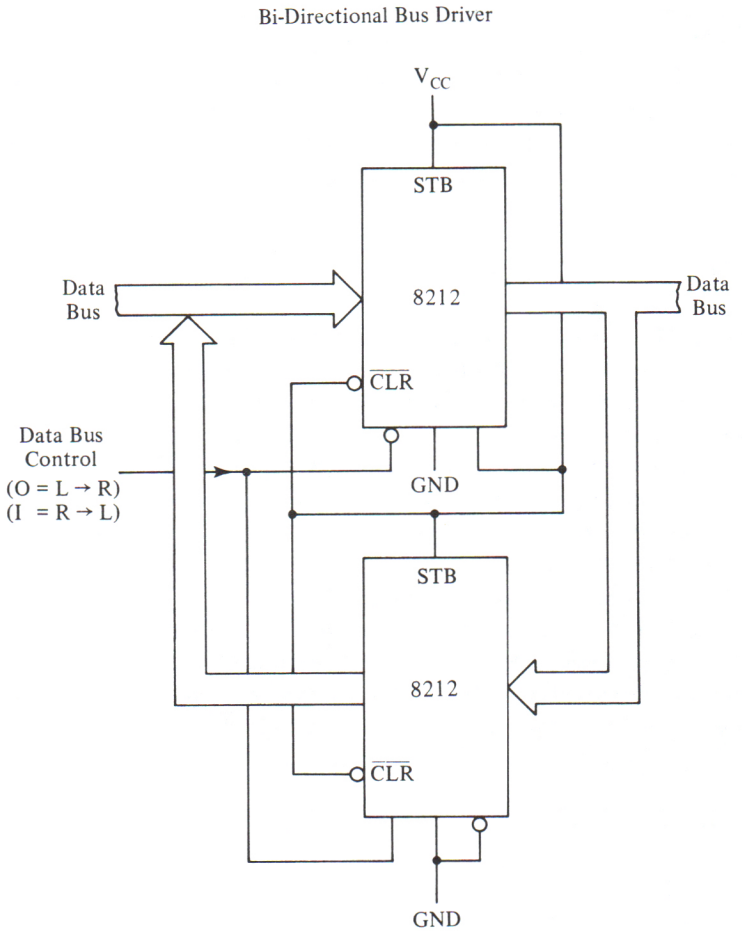


FIGURE 6-10(b). Connection of the 8212.

active-LOW terminal that controls the tri-state outputs. If $\overline{CS}$ is HIGH, then the outputs have a high impedance and will not load the bus. However, when $\overline{CS}$ is LOW, the output lines become active.

Only the $\overline{DIEN}$ line of the 8216/8226 devices affects the input function. To make an output port active, though, requires a HIGH on $\overline{DIEN}$ *and* a LOW on $\overline{CS}$.

The actual circuit configuration will depend upon the use of the chip (I or 0) and the microprocessor used. In a Z80 system, for example, we could connect $\overline{CS}$ to $\overline{IORQ}$ and $\overline{DIEN}$ to $\overline{RD}$ (which will be LOW for input operations and HIGH for output operations).

The Intel 8212 device is shown in Figure 6-10(a). This chip is an eight-bit directional I/O port that was designed to be compatible with the 8080A

microprocessor. It also has been used with many other microprocessor chips. The 8212 also differs from the previous example in that it is (a) unidirectional (two are needed for a complete I/O), and (b) it contains latches so it will retain the last valid data. The latter feature is necessary for most microcomputer applications. An example showing a pair of 8212 devices in a bidirectional I/O port circuit is given in Figure 6-10(b).

SERIAL I/O PORTS

Parallel I/O ports prove to be expensive when data must be communicated over a long distance to another device. Unless the communications link is only a few feet in length, serial communications is cheaper. Transmission of eight data bits in parallel, for example, requires at least eight lines, and usually a ninth line is needed as a common or ground. The excessive cost of radio and telephone links should be obvious. In contrast, a serial communications link requires only one channel, be it local wire, telephone lines, or a radio channel.

In addition, there are numerous peripherals, such as teletypewriters, that use serial inputs or provide serial outputs. Serial data ports are slower than parallel ports because the data must be transmitted one bit at a time. The bit length is also greater because certain *format bits* must be added, i.e., start bit plus one or more stop bits.

SOFTWARE SERIAL I/O PORTS

Some designers use software to force a single bit of a parallel I/O port to function as if it were a serial port. In one simple method, the output word is loaded into the accumulator, and then the *shift* instructions are used to output the word bit by bit.

Another, more formal, technique is to write a software program to perform the logic functions of a *universal asynchronous receiver/transmitter,* or UART. Such programs are often designated *teletypewriter* or *printer driver routines.*

TTL/CMOS Hardware Methods

The basic circuit needed to make a serial output port is the *parallel-in serial-out* (PISO) shift register. Data from the accumulator is written to the parallel inputs of the shift register as if the register were either a memory location or a parallel output port. A program is then executed that will clock

the flip-flop left to right, causing the shift register to output the data in serial format. Various clocking schemes are used, including successively generating an OUT signal, or writing to a specific (unused) memory location. In both cases, a device or location select pulse must be provided.

A serial input port can be built in a similar manner, but using a *serial-in parallel-out* (SIPO) shift register. Remember, tri-state outputs must be available to prevent loading of the data bus when the port is inactive.

The UART

Until recently the design of serial I/O ports was difficult and was considered at best a nettlesome chore for the digital design engineer. But today we have a specialized class of LSI integrated circuits that will do the job: the universal asynchronous receiver/transmitter, or UART.

Synchronous transmission requires that the clock pulses at both ends of the circuit be matched because data is transmitted at precise times. In general, the respective baud rates of the receiver and transmitter must be within ±0.01 percent of each other. Such precision adds complexity and cost to the system. We can reduce both complexity and cost, however, by using *asynchronous* transmission.

Figure 6-11 shows the block diagram for the now-standard TR1602A/B device. This UART is essentially the same as the popular AY-1013 UART device. It is capable of either full duplex or half duplex operation because the receiver and transmitter sections are totally independent of each other, except for power supply connections.

The UART chip is partially useful because it can be programmed externally for several different *bit lengths, baud rates, parity* (odd-even, receiver verification/transmitter generation), *parity inhibit,* and *stop bit* length (1, 1.5, or 2 stop bits). The UART also provides six different status flags: *transmission completed, buffer register transfer completed, received data available, parity error, framing error,* and *overrun error.*

The clock speed is 320 kHz (maximum) for the A and B versions, 480 kHz for the AO3/BO3 versions, 640 kHz for the AO4/BO4 versions, and to 800 kHz for the AO5/BO5 series. The receiver output lines are tri-state logic, so will float at a high impedance to both ground and the +5-volt line when inactive. The use of tri-state output allows the device to be connected directly to the data bus of a computer or other digital instrument.

The transmitter section uses an eight-bit parallel input register that will accept data to be sent serially. It will convert the eight-bit data word received in the input register to serial format that includes the eight-bit word (also formatable to 5, 6, or 7 bits), start bit, parity bit and stop bits.

The receiver can be viewed as simply the mirror image of the

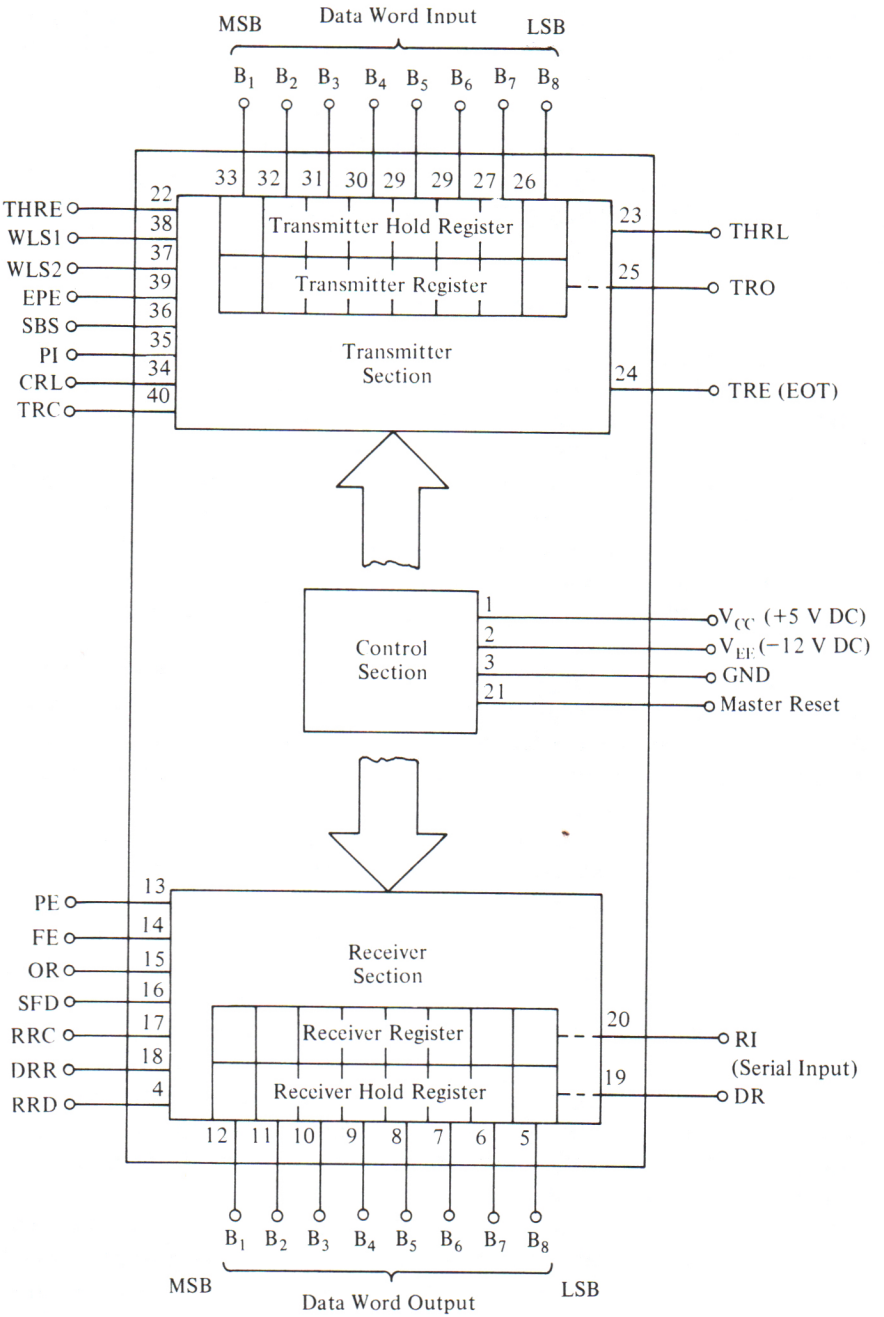


FIGURE 6-11. Block diagram of a UART.

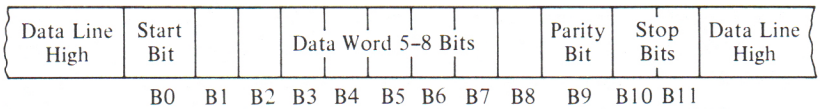


FIGURE 6-12. UART data word format.

transmitter. It receives a serial input word containing start bits, data, parity, and stop bits. This serial stream of data is checked for validity by comparison with parity and for the existence of the stop bits.

The UART data format (serial) is shown in Figure 6-12. The transmitter output pin will remain HIGH unless data is being transmitted. Start bit BØ is always a HIGH-to-LOW transition, which tells the system that a new data word is about to be sent. Bits B1 through B8 are the data bits loaded into the transmitter on the sending end of the system. All eight bits of the maximum word length format are shown in the figure, even though truncated word lengths of 5, 6, or 7 bits are also allowable. The stop bit length can be programmed for 1, 1.5, or 2 bits, according to the needs of the system designer.

The number of data bits, the parity, and the number of stop bits are programmed onto the device using HIGH and LOW levels applied to certain pins designated for that purpose. For example, the WLS1 and WLS2 pins are used as *word length select* pins, and will set the data word length according to the following code system:

WORD LENGTH	WLS1	WLS2
5 bits	0	0
6 bits	1	0
7 bits	0	1
8 bits	1	1

Similarly, a two-bit stop code is selected by connecting SBS HIGH, but only when the data word is six, seven, or eight bits in length. If the data word is set to five bits length, which is used on Baudot teletypewriters, then the 1.5 bit stop code is used. If SBS is LOW, then the stop code is one bit in length. The parity is set by the EPE pin, and will be coded *odd* for a LOW and *even* for a HIGH.

The clock on a UART system must be stable, so we cannot generally use RC timer-based clocks and expect proper performance, especially at higher baud rates. The frequency of the clock must be 16 times the baud rate. If we want to transmit data at 300 baud, for example, the oscillator frequency must be 300 × 16, or 4800 Hz. While this frequency is well within the range normally competent RC oscillators can produce, it is recommended that a

crystal oscillator be used to ensure the stability and accuracy of the clock. An attractive alternative is the CMOS 4060 device, which contains an internal crystal or RC oscillator and a chain of binary divider stages.

The transmitter circuit is shown in Figure 6-13(a). Note that only the eight-bit input data, clock and serial output are required to make this circuit operational. The TRE, THRE and THRL signals are status flags, and are optional although they will probably be used in most practical cases. They convey information about the status of the information transfer, and these are sometimes needed in the software used to control the UART. A careful review of the meaning of each flag is necessary for those designers who wish to use the UART.

The basic receiver circuit for the UART is shown in Figure 6-13(b). We have a similar simplicity in the receiver section (one of the principal attractions held by LSI devices to equipment and instrument designers). Only the clock, serial input, and eight-bit parallel output lines are needed. Again, however, certain signals are available that will make some applications either easier or possible. These are the DR, OE, FE, and PE flags. You can look up the meanings of these signals, and those of the transmitter section, in Table 6-1.

Notice in the receiver section that we use an inverter from the *data received* terminal to reset the DRR terminal. This signal tells the UART to get ready for the next character and can be used to signal a distant transmitter that the UART is ready to receive another transmission.

One of the things about the UART that appeals to many designers is that the two sections (receiver and transmitter) can be used either independently or in a common system. In a simplex communications channel (one direction only), a transmitter-wired UART is used on the transmitter end, while a receiver-wired UART is used on the receiving end. In a half-duplex system (bidirectional communication, but only one direction at a time), both sections are used at each end, and the status flags can be used in a handshaking system to coordinate matters. Full-duplex operation is possible, but requires either a second channel (especially in radio links) or a second set of audio tones in hard-wired telephone line systems. Not all telephone lines, however, are amenable to full duplex operation, especially over long distance lines.

In dedicated instrument applications, the programming pins will probably be hard-wired in the proper codes, but in many cases switches are used to allow the user to program as needed. You can also connect the UART control pins to an I/O port to permit programming of the UART under software control of the computer. This method, incidentally, is behind some of the programmable LSI I/O port devices that are designed to work with specific microcomputers. Such a system allows the user of the computer to "custom design" the communications port according to immediate program needs.

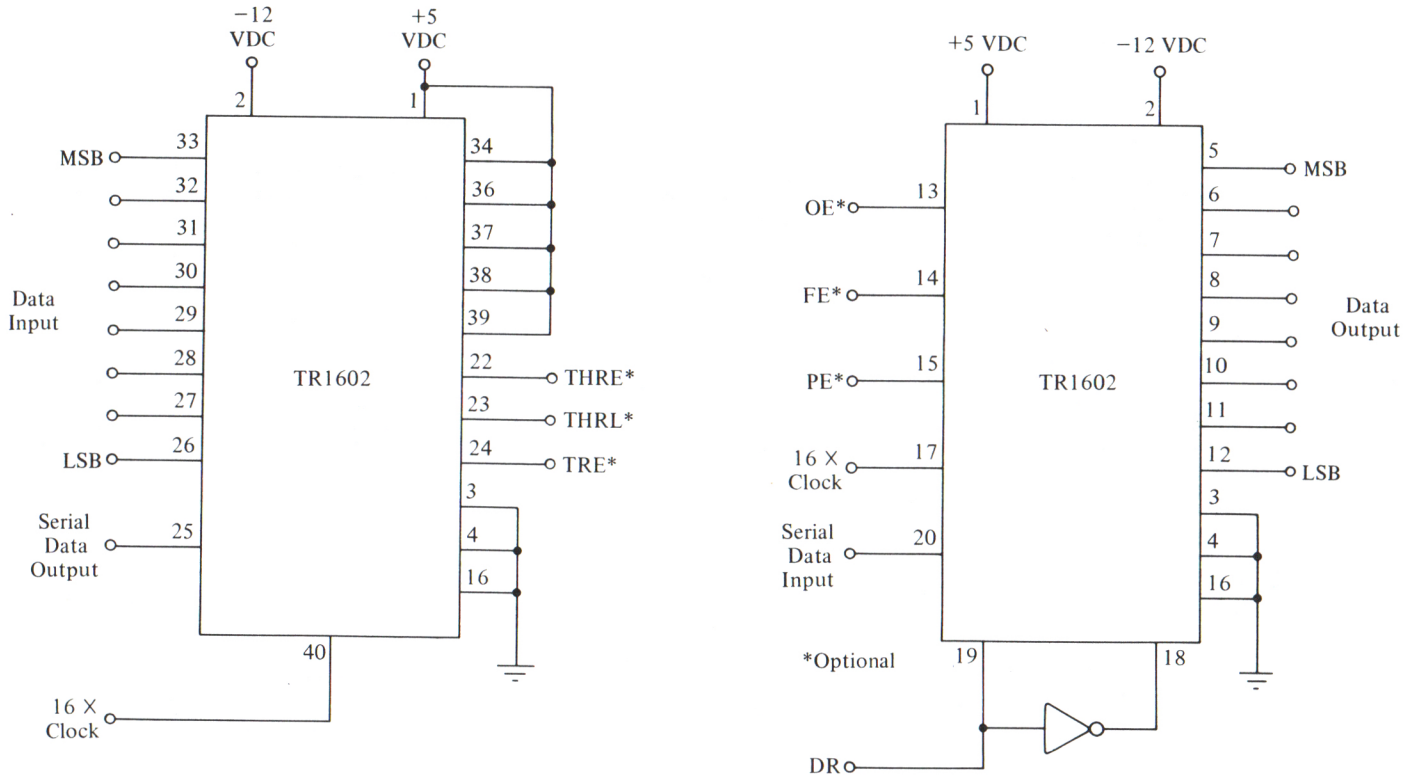


FIGURE 6-13. (a) TR1602 UART — transmitter connections. (b) Receiver connections for the TR1602 UART.

Table 6-1

PIN NO.	MNEMONIC	FUNCTION
1	V_{CC}	+5 volts DC power supply.
2	V_{EE}	−12 volts DC power supply.
3	GND	Ground.
4	RRD	Receiver Register Disconnect. A high on this pin disconnects (i.e., places at high impedance) the receiver data output pins (5 through 12). A low on this pin connects the receiver data output lines to output pins 5 through 12.
5	RB_8	LSB — Receiver data output lines
6	RB_7	
7	RB_6	
8	RB_5	
9	RB_4	
10	RB_3	
11	RB_2	
12	RB_1	MSB
13	PE	Parity error. A high on this pin indicates that the parity of the received data does not match the parity programmed at pin 39.
14	FE	Framing Error. A high on this line indicates that no valid stop bits were received.
15	OE	Overrun Error. A high on this pin indicates that an overrun condition has occurred, which is defined as not having the DR flag (pin 19) reset before the next character is received by the internal receiver holding register.
16	SFD	Status Flag Disconnect. A high on this pin will disconnect (i.e., set to high impedance) the PE, FE, OE, DR, and THRE status flags. This feature allows the status flags from several UARTs to be bus-connected together.
17	RRC	16 × Receiver Clock. A clock signal is applied to this pin, and should have a frequency that is 16 times the desired baud rate (i.e., for 110 baud standard it is 16 × 110 baud, or 1760 hertz).
18	DRR	Data Receive Reset. Bringing this line low resets the data received (DR, pin 19) flag.
19	DR	Data Received. A high on this pin indicates that the entire character is received, and is in the receiver holding register.

Table 6-1 (*Continued*)

PIN NO.	MNEMONIC	FUNCTION
20	RI	Receiver Serial Input. All serial input data bits are applied to this pin. Pin 20 must be forced high when no data is being received.
21	MR	Master Reset. A short pulse (i.e., a strobe pulse) applied to this pin will reset (i.e., force low) both receiver and transmitter registers, as well as the FE, OE, PE, and DRR flags. It also sets the TRO, THRE, and TRE flags (i.e., makes them high).
22	THRE	Transmitter Holding Register Empty. A high on this pin means that the data in the transmitter input buffer has been transferred to the transmitter register, and allows a new character to be loaded.
23	THRL	Transmitter Holding Register Load. A low applied to this pin enters the word applied to TB1 through TB8 (pins 26 through 33, respectively) into the transmitter holding register (THR). A positive-going level applied to this pin transfers the contents of the THR into the transmit register (TR), unless the TR is currently sending the previous word. When the transmission is finished the THR → TR transfer will take place automatically even if the pin 25 level transition is completed.
24	TRE	Transmit Register Empty. Remains high unless a transmission is taking place, in which case the TRE pin drops low.
25	TRO	Transmitter (Serial) Output. All data and control bits in the transmit register are output on this line. The TRO terminal stays high when no transmission is taking place, so the beginning of a transmission is always indicated by the first negative-going transition of the TRO terminal.
26	TB_8	LSB } Transmitter input word.
27	TB_7	
28	TB_6	
29	TB_5	
30	TB_4	
31	TB_3	
32	TB_2	
33	TB_1	MSB
34	CRL	Control Register Load. Can be either wired permanently high, or be strobed with a positive-going pulse. It loads the programmed instructions (i.e., WLS1, WLS2, EPE,

Table 6-1 (*Continued*)

PIN NO.	MNEMONIC	FUNCTION
		PI, and SBS) into the internal control register. Hard wiring of this terminal is preferred if these parameters never change, while switch or program control is preferred if the parameters do occasionally change.
35	PI	Parity Inhibit. A high on this pin disables parity generator/verification functions, and forces PE (pin 13) to a low logic condition.
36	SBS	Stop Bit(s) Select. Programs the number of stop bits that are added to the data word output. A high on SBS causes the UART to send two stop bits if the word length format is 6, 7, or 8 bits, and 1.5 stop bits if the 5-bit teletypewriter format is selected (on pins 37-38). A low on SBS causes the UART to generate only one stop bit.
37 38	WLS_1 WLS_2	Word Length Select. Selects character length, exclusive of parity bits, according to the rules given in the chart below: Word Length — WLS1 — WLS2 5 bits — low — low 6 bits — high — low 7 bits — low — high 8 bits — high — high
39	EPE	Even Parity Enable. A high applied to this line selects even parity, while a low applied to this line selects odd parity.
40	TRC	16 × Transmit Clock. Apply a clock signal with a frequency that is equal to 16 times the desired baud rate. If the transmitter and receiver sections operate at the same speed (usually the case), then strap together TRC and RRC terminals so that the same clock serves both sections.

Z80-PIO

The Z80-PIO Zilog IC (also called the Mostek MK3881) is a special parallel I/O controller. It is a programmable, two-port I/O device. It contains two independent, bidirectional, eight-bit ports for data transfer between the Z80 CPU chip and external devices. Complete "handshake" capability is permitted. There are four distinct modes of operation in the Z80-PIO: *byte*

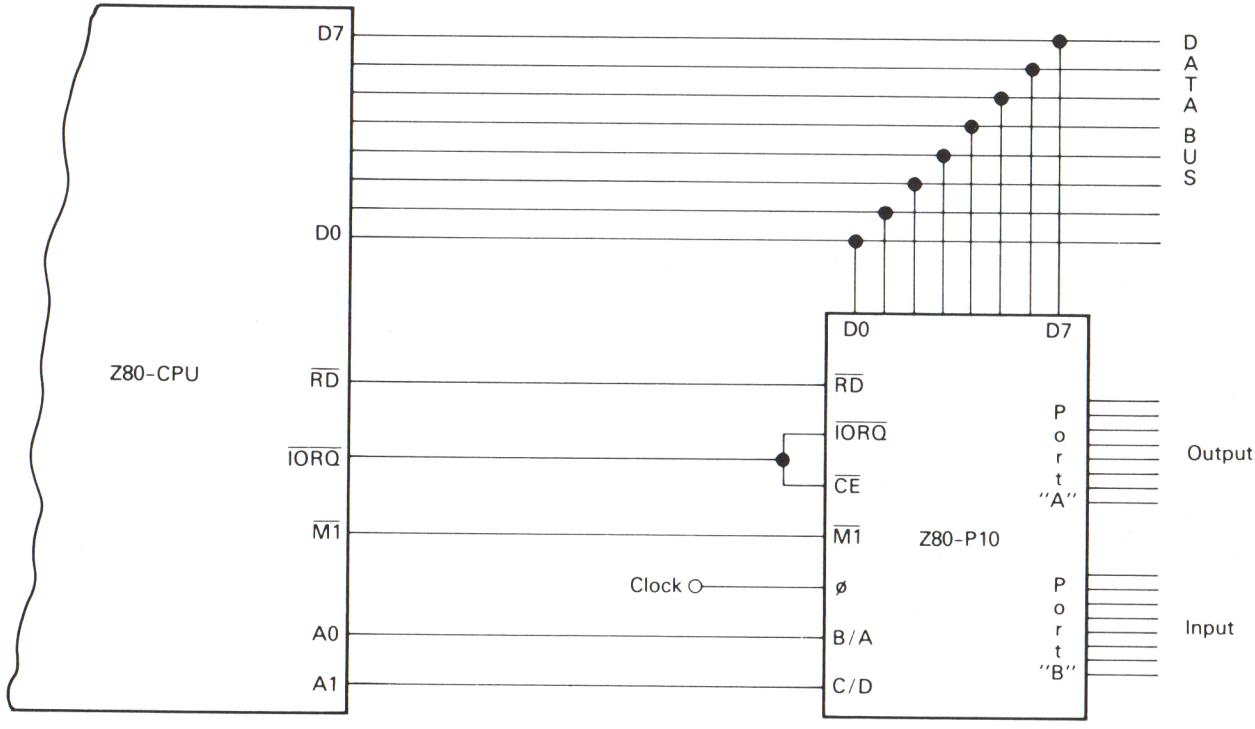


FIGURE 6-14. Interfacing the Z80-PIO device for I/O function.

output, byte input, byte bidirectional bus (port A only), and *bit control.* Only a single +5-volt dc power supply and a single-phase clock are required.

One of the more powerful aspects of the Z80-PIO is that it permits use of the interrupt modes of the Z80 CPU chip. The Z80-PIO is semi-intelligent, in that it can be programmed to recognize specific interrupts (i.e., alarms) from external devices. Zilog bills this feature as allowing the CPU to spend time on chores other than polling the external device for an interrupt condition.

Figure 6-14 shows a simple I/O port using the Z80-PIO IC. The DØ-D7 inputs of the PIO are connected to their respective lines on the data bus. Similarly, the $\overline{\text{IORQ}}$ and $\overline{\text{M1}}$ inputs on the PIO are connected to their respective outputs on the Z80 IC. The $\overline{\text{IORQ}}$ and $\overline{\text{CE}}$ (i.e., chip enable) inputs of the PIO are connected to the $\overline{\text{IORQ}}$ output on the Z80. In this example, we have designated PIO port-A as the output port, and PIO port-B as the input bidirectional I/O ports by appropriate programming of the B/A pin.

The Z80-SIO is a serial communications controller that can be directly interfaced with the Z80 CPU chip. This device is very similar to the UART IC, except that it permits both synchronous and asynchronous communications, and has two channels (A and B).

CHAPTER 7

INTERFACING MEMORY TO THE Z80

The Z80 microprocessor chip uses a 16-bit address bus, so it is able to directly address up to 2^{16}, or 65,536 memory locations. Note that this upper limit is usually written "64K" rather than "65K" because a "computer-K" is 1024 instead of 1000. the Z80 data bus uses one byte (eight bits), so each memory location can store a single eight-bit word.

The mixture of possible memory devices used with the Z80 includes static random access memory (RAM), dynamic RAM, read only memory (ROM), programmable read only memory (PROM) erasable PROM (EPROM), plus a number of devices such as analog-to-digital converters (ADC), and digital-to-analog converters (DAC), which are sometimes treated as memory. This technique, called *memory mapping,* makes some data acquistion chores easier (or at least faster).

CONTROL SIGNALS FOR MEMORY OPERATIONS

We must be cognizant of the basic Z80 control signals that apply to memory operations: $\overline{\text{MREQ}}$, $\overline{\text{WR}}$, and $\overline{\text{RD}}$. These signals are the memory request, write, and read, respectively. The memory request signal will drop LOW whenever the CPU is executing either a memory read or memory write operation. It tells the system that the data on the bus are memory data. If a

memory write operation is taking place, then the *write* ($\overline{WR}$) signal will also go LOW. If, on the other hand, it is a memory read, then the *read* ($\overline{RD}$) signal will go LOW. All memory operations, therefore, will generate LOW on two control pins of the Z80: $\overline{MREQ}/\overline{WR}$ for memory write operations and $\overline{MREQ}/\overline{RD}$ for memory read operations.

Most integrated circuit memory devices have at least one chip enable (CE) pin, and some have two chip enable pins (labelled CE1 and CE2). There also may be a *read/write* (W/R) pin to instruct the device whether the desired operation is a memory read or a memory write.

One of the simplest cases is shown in Fig. 7-1. Here we see 1024 bytes of *read only memory* (ROM) interfaced directly to the Z80. In this case, we have assigned the ROM to the lower 1K of the memory address range. The locations available, then, are 00 00 (H) to 03 FF (H). Since we are dealing with the lower 1K, we need only the lower-order byte of the address bus, AØ–A7, plus the two least significant bits of the upper-order byte (A8 and A9).

Two chip enable (CE) terminals are available. We use one of them (CE2) to make sure that the ROM will respond only to addresses in the lower 1K of memory. Address bus bit A10 will always remain LOW when the CPU is addressing a location in the lower 1K, but will go HIGH when an address greater than 03 FF (H) is selected. The ROM, therefore, is enabled only when the address on the address bus is less than 03 FF (H).

The second chip enable pin (CE1) is used to turn on the ROM only when the memory read operation is taking place. This CE pin wants to see a HIGH for turn-on of the ROM. Recall that a NOR gate will output a HIGH only when both inputs are LOW. We can, therefore, create a device select command for CE1 by applying the $\overline{MREQ}$ and $\overline{RD}$ control signals from the CPU to the inputs of a NOR gate. CE1 will go HIGH, then, only when a memory read operation takes place.

At least two of the more popular ROM chips require only a single chip enable command. In the example shown in Fig. 7-2(a), the chip enable is an active-LOW input (so is designated $\overline{CE}$). This terminal is brought LOW

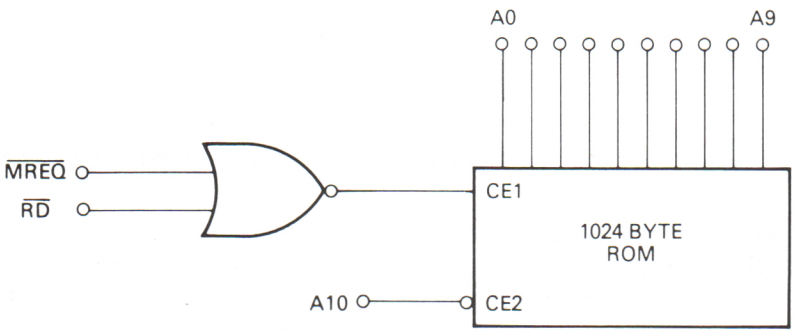
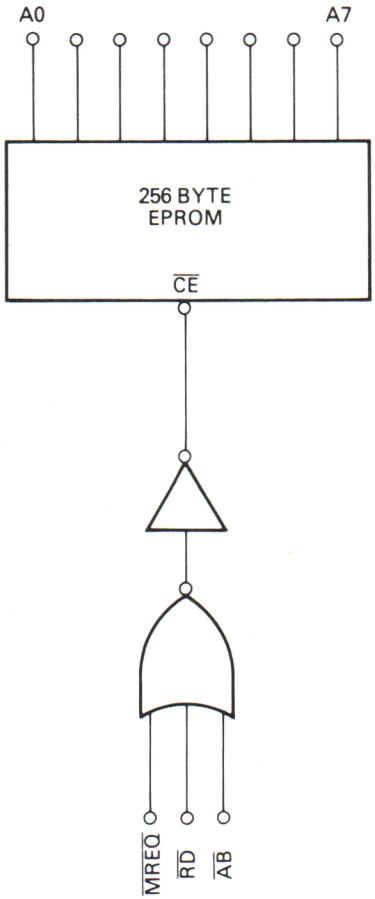
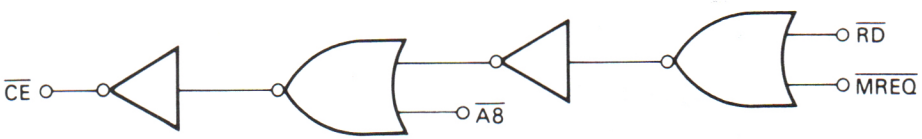


FIGURE 7-1. Operating CE of memory from MREQ and RD.



(a)



(b)

FIGURE 7-2. (a) Enabling EPROM from MREQ/RD/A8; (b) Same function accomplished with two-input gates.

whenever we want to read the contents of one of the locations in the chip.

The example shown in Fig. 7-2(a) is a 256-byte ROM, with a single $\overline{CE}$ terminal. We must, therefore, construct external circuitry that will bring the chip enable terminal LOW when we want to perform the read operation. The simplest way is to use a three-input NOR gate and an inverter. The output of

the NOR gate will go HIGH only when all three of the inputs are LOW. We connect the $\overline{MREQ}$, $\overline{RD}$, and bit A8 of the address bus to the respective inputs of the NOR gate. When the conditions are met, then the output of the gate snaps HIGH, and is then inverted to become the $\overline{CE}$ signal required by the EPROM chip.

An alternative method is shown in Fig. 7-2(b). Here we are using two inverters and a pair of NOR gates to form the $\overline{CE}$ signal. The idea is to cause $\overline{CE}$ to go LOW when the three conditions are met. To do this, we must see both inputs of NOR gate G2 LOW simultaneously. One of the inputs is connected to bit A8 of the address bus, while the other is connected to the inverted output of NOR gate G1. The inputs of G1 are, in turn, connected to $\overline{MREQ}$ and $\overline{RD}$ signals.

A situation that is a little more complicated is shown in Fig. 7-3. Here we are interfacing static RAM devices that have a chip enable and a $\overline{R}/W$ terminal. This latter terminal will cause the device to read out data when LOW, and allow writing in data when HIGH. We connect the $\overline{R}/W$ terminal, then, to the $\overline{RD}$ signal of the Z80 CPU.

The chip enable in this example wants to see a HIGH in order to turn on the device. We can, then correct CE to the output of a NOR gate. The $\overline{MREQ}$ and $\overline{A8}$ signals are connected to the two inputs of the NOR gate. If both of these signals go LOW simultaneously, and the $\overline{RD}$ is also LOW, a memory read operation takes place from the location addressed by AØ–A7.

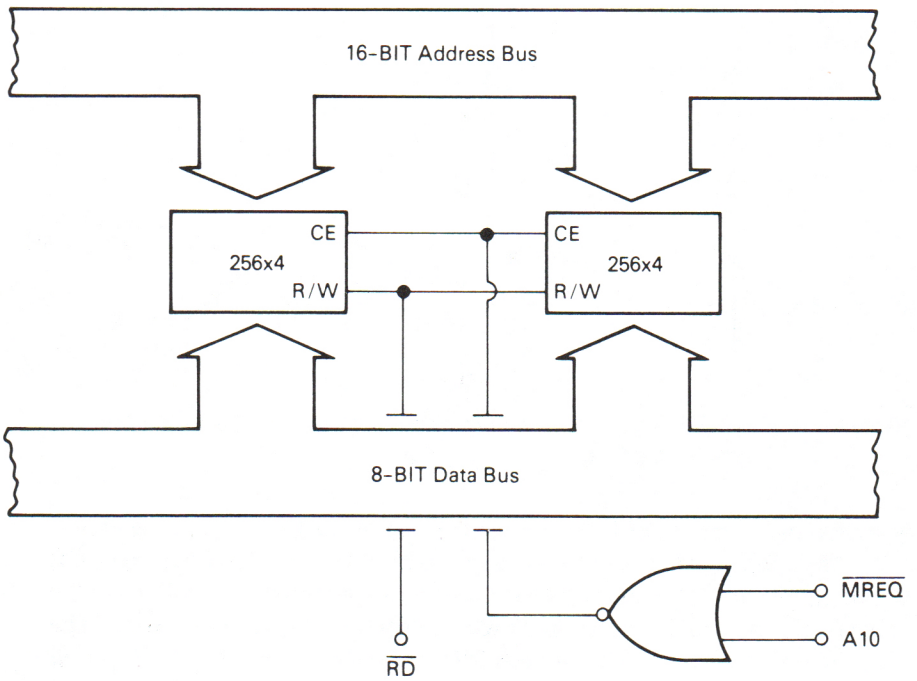


FIGURE 7-3. Interfacing RAM.

Alternatively, if the $\overline{MREQ}$ and $\overline{A7}$ signals are LOW, and the $\overline{RD}$ signal is HIGH, then a memory write operation will take place.

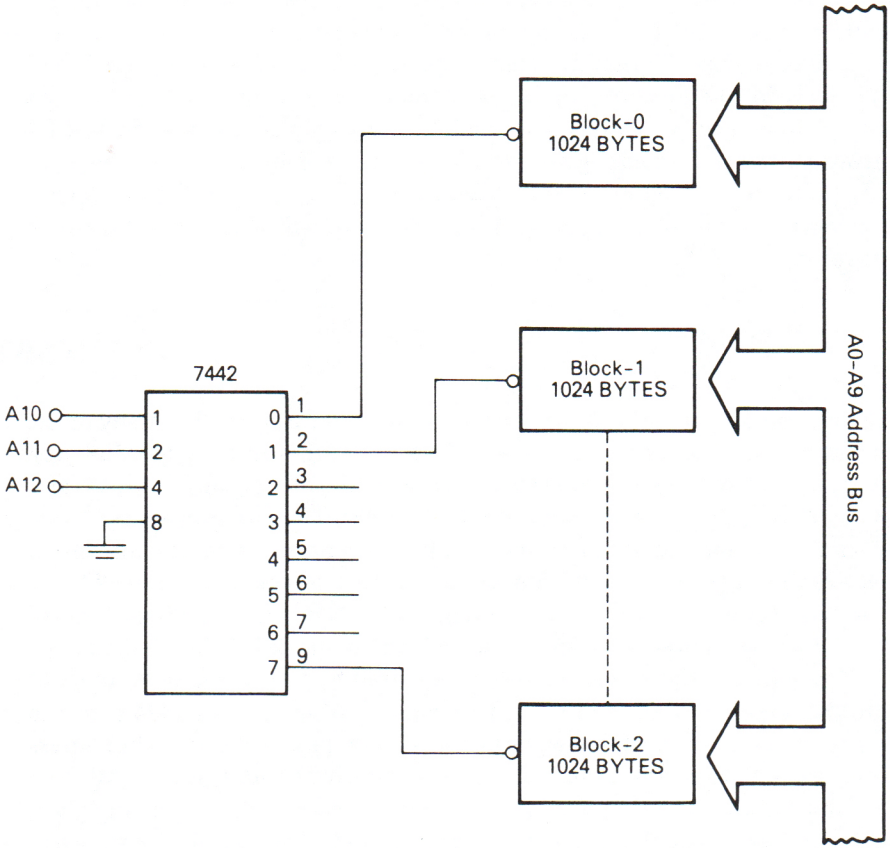
Note in Fig. 7-3 that two chips are used to form a 256-byte static RAM memory. Most memories require more than a single chip in order to form a complete byte-array. In this case, each memory chip contains a 256 × 4-bit array, so two connected together will form a 256 × 8-bit array (i.e., 256 bytes of memory). The popular 2102 device is listed as a 1024 × 1-bit device. Connecting eight of these devices into an array will result in a 1024-byte memory.

ADDRESS DECODING

In most microcomputers more than 1K of memory is used. But many of the memory chips available are only 1024-byte (with some being 256-byte). Although there are more modern devices capable of very large byte arrays, many users still prefer the older, smaller devices. The question arises, "How does the memory device allocated to a location greater than the maximum address in each individual chip know when it is being addressed?" The solution seems to be ordering of the memory in 1K blocks, and then the use of some form of address decoding to tell which 1K block is being designated.

Figure 7-4 shows a selection scheme used by several manufacturers of 8K memory banks. Each block of this memory is an array of 1024 bytes, so every location can be addressed by bits AØ–A9 of the address bus. The address pins for all devices are connected together to form the address bus (AØ–A9). We must, however, select which of the eight blocks is addressed at any given time. One way to do this is to use a data selector IC. The 7442 device shown in Fig. 7-4 is a BCD-to-one-of-ten decoder. It will examine a four-bit binary (i.e., BCD) input word, and issue an output condition that indicates the value of that word. In this simplified example, we are going to limit the memory size to 8K, so only the 1, 2, and 4 inputs of the 7442 are needed. The input weighted 8 is grounded (i.e., set = Ø). The 7442 indicates the active output by going LOW, exactly the right condition for the RAM devices in the memory blocks. The table below shows the code that will exist on the A1Ø–A12 bits of the address bus for the various memory addresses in the range 0–8K.

MEMORY LOCS.	A13	A12	A11	A10	BLOCK NO.	7442 OUTPUT	7442 PIN
0K–1K	0	0	0	0	0	0	1
1K–2K	0	0	0	1	1	1	2
2K–3K	0	0	1	0	2	2	3
3K–4K	0	0	1	1	3	3	4
4K–5K	0	1	0	0	4	4	5
5K–6K	0	1	0	1	5	5	6
6K–7K	0	1	1	0	6	6	7
7K–8K	0	1	1	1	7	7	9



(a)

		A13	A12	A11	A10
Block-Ø	0-1K	Ø	Ø	Ø	Ø
	1K-2K	Ø	Ø	Ø	1
	2K-3K	Ø	Ø	1	Ø
	3K-4K	Ø	Ø	1	1
	4K-5K	Ø	1	Ø	Ø
	5K-6K	Ø	1	Ø	1
	6K-7K	Ø	1	1	Ø
	7K-8K	Ø	1	1	1

(b)

FIGURE 7-4. (a) Using 7442 in bank selection of memory; (b) Code for above.

For an 8K memory, then, the lower 10 bits of the address bus (AØ–A9) select which location in the individual chips is wanted, and A10–A12 select which block of 1024 bytes contains the address.

In the example of Fig. 7-4 we limited the memory size to a mere 8K. This was done intentionally to keep the circuit simple. But how do we select memory in ranges higher than 8K? The answer is to use the 7442 input weighted "8" as a bank select control. Recall from Fig. 7-4 that this input was kept grounded. If it is HIGH, then none of the eight outputs of the 7442 will go LOW. But if it is LOW, then the circuit will work. Figure 7-5 shows a simplified selection scheme for all 65K addressable by the Z80, using the "8" weighted inputs of the 7442 block selectors as a *bank select* terminal. Each bank of 8K contains its own block select 7442, and one additional 7442 is used to select the bank of 8K that will become active. The table below shows the codes existing on address lines A13–A15 for each 8K bank of locations:

MEMORY LOCS.	BANK NO.	A15	A14	A13	7442 OUTPUT	7442 PIN LOW
0K–8K	Ø	Ø	Ø	Ø	Ø	1
8K–16K	1	Ø	Ø	1	1	2
16K–24K	2	Ø	1	Ø	2	3
24K–32K	3	Ø	1	1	3	4
32K–40K	4	1	Ø	Ø	4	5
40K–48K	5	1	Ø	1	5	6
48K–56K	6	1	1	Ø	6	7
56K–64K	7	1	1	1	7	9

Figure 7-6 shows an alternative bank selection circuit that is based on a three-input NAND gate (i.e., one section of a 7410 TTL IC device). The properties of a NAND gate are

1. If any input is LOW, the output is HIGH.
2. If *all* inputs are HIGH, then the output is LOW.

In this case, then, all three of the inputs must be HIGH for the output to drop LOW. If the output of the NAND gate is used to drive the "8" input of the 7442, then the particular bank served by that 7442 will be selected only when all three inputs are HIGH.

How do we contrive the circuit to force all inputs HIGH only when the correct bit pattern is seen on lines A13–A15? The solution is the inverters and switches shown in Fig. 7-6. Each switch selects either the inverted (i.e., "Ø" position) or noninverted (i.e., "1" position) versions of each address bus signal. We set the three switches according to our bank selection format, using the codes from the table given previously. Each switch is to be set to the

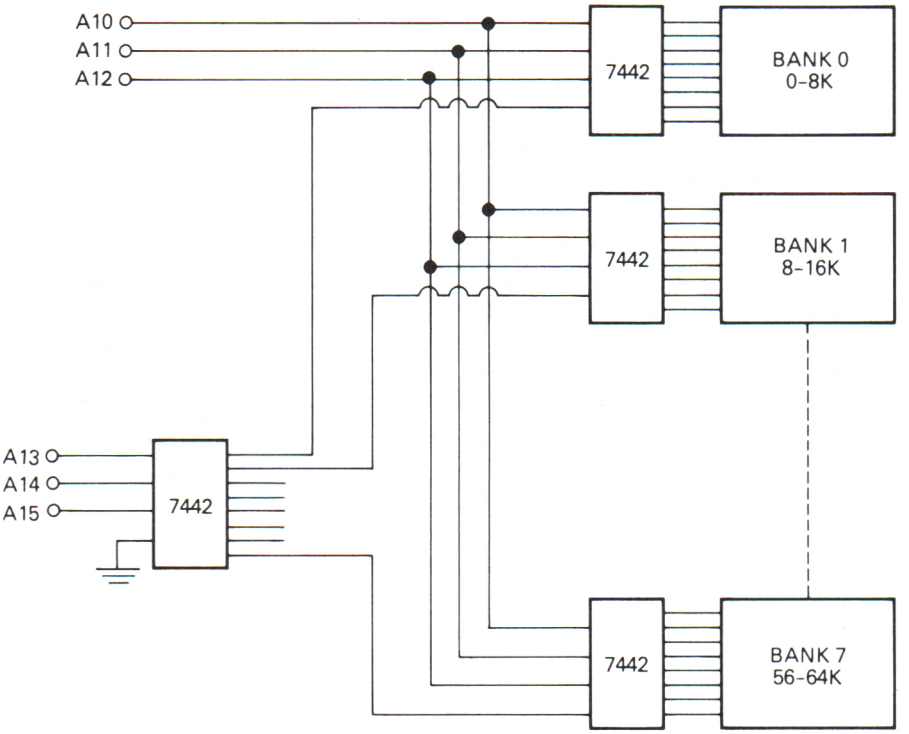


FIGURE 7-5. Multiple bank selection.

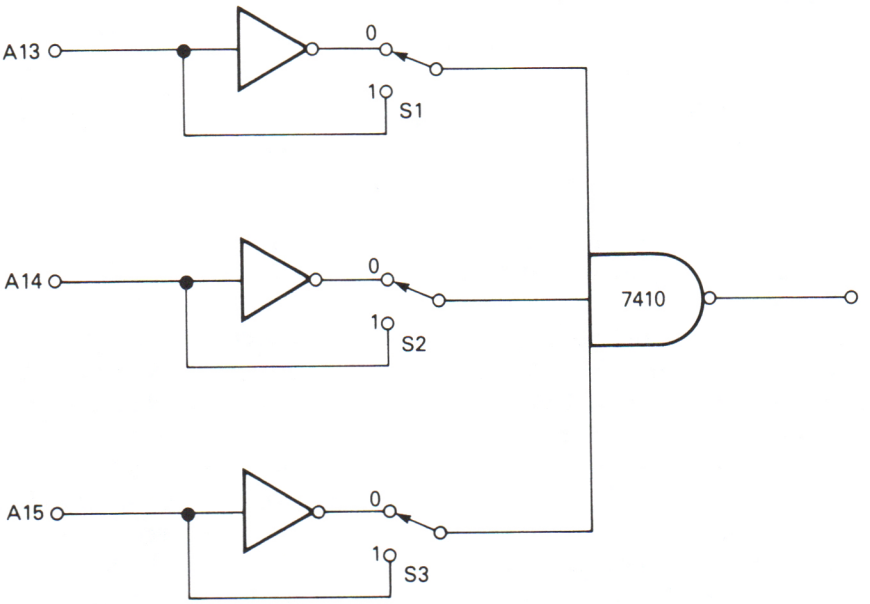


FIGURE 7-6. Switches determine whether logical-1 or logical-0 is required.

position corresponding to the digit expected at that input when the address bus code is correct. For bank Ø, for example, the code is ØØØ. If S1–S3 are set to "Ø" position, then the NAND gate sees the inverted address line signals. When ØØØ appears on A13–A15, the NAND gate sees 111. Since this is the condition required, the output drops LOW and turns on the selected bank.

Note that Intel manufacturers a 1-of-8 decoder intended specifically for bank selection in the 8080A device. It should also work nicely with the Z80.

DYNAMIC MEMORY

Dynamic memory (RAM) will not hold its data for an indefinite length of time, unless a refresh operation is performed. The refresh operation is a function of the CPU in most cases, although some non-CPU examples exist. Although the use of static RAM will eliminate this problem, it is only at the cost of a higher power consumption. The Z80A device provides for refresh of the dynamic memory by adding a refresh segment to the M1 (instruction fetch) machine cycle.

During clock periods T3 and T4 of the M1 cycle, used by the Z80 for the decoding of the instructions fetched in the earlier T-periods, a refresh signal is generated. The $\overline{\text{RFSH}}$ terminal (pin 28) of the Z80 will go LOW during this period. Note that this signal must be used in conjunction with the $\overline{\text{MREQ}}$ (memory request) signal, bacause the $\overline{\text{RFSH}}$ is guaranteed to be stable only when the $\overline{\text{MREQ}}$ is also active.

During the refresh period the lower portion of the address of a refresh location is placed on the lower seven bits (AØ–A6) of the address bus (A7 is Ø). The data on AØ–A6 are from the R register in the Z80, which is incremented after each instruction fetch. The upper eight bits of the address bus carry the contents of the I register. Figure 7-7 shows an example of an 8K dynamic RAM interfaced to a Z80. In this particular case, 4K × 8-bit dynamic RAMs are used. If no other RAM is used, we may use bit A12 of the address bus as a chip-select line.

ADDING WAIT STATES

All solid-state memory chips require a certain minimum period of time to write data into, or read data from, any given location. Many such devices are graded (and priced!) according to memory speed. The popular 2102 device, a 1K × 1-bit IC, is available in 250-nanosecond, 400-nanosecond, and 500-nanosecond versions. Of course, the cost per chip rises with the speed.

Since the Z80 A can operate at speeds up to 4mHz, we sometimes find the cycle (M1 or memory) over before the data have settled to, or from, memory.

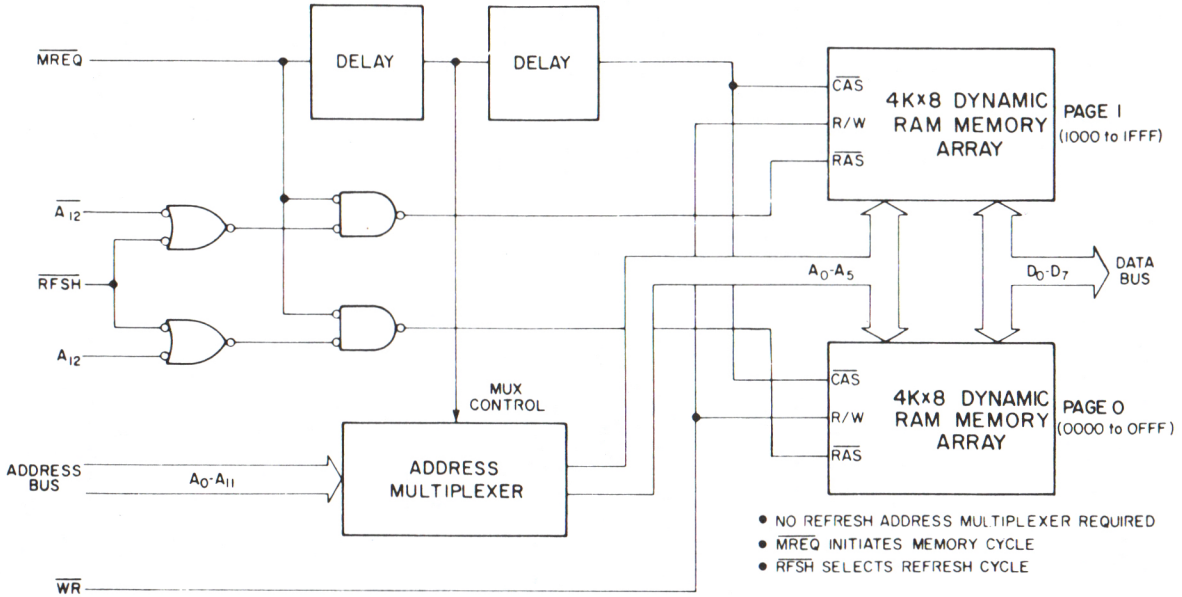

MREQ
DELAY
DELAY
A12
RFSH
A12
MUX CONTROL
ADDRESS BUS
A0-A11
ADDRESS MULTIPLEXER
WR
CAS
R/W
RAS
4K×8 DYNAMIC RAM MEMORY ARRAY
PAGE 1
(1000 to 1FFF)
A0-A5
D0-D7
DATA BUS
CAS
R/W
RAS
4K×8 DYNAMIC RAM MEMORY ARRAY
PAGE 0
(0000 to 0FFF)
• NO REFRESH ADDRESS MULTIPLEXER REQUIRED
• MREQ INITIATES MEMORY CYCLE
• RFSH SELECTS REFRESH CYCLE

FIGURE 7-7.

This problem can be overcome by adding the circuitry shown in Fig. 7-8. Both of these circuits generate a $\overline{\text{WAIT}}$ input (pin 24 of the Z80) equal to the period of one clock pulse.

The circuit in Fig. 7-8(a) uses both sections of a TTL 7474 dual Type-D flip-flop. The 7474 is a positive-edge triggered device, meaning that data on the D input are transferred to the Q output *only* during the positive-going transitions of the clock pulse.

Immediately after the onset of clock pulse T1, the $\overline{\text{M1}}$ line goes LOW, forcing the D input of FF1 LOW. When clock pulse T2 snaps HIGH, then, this LOW is transferred to the Q output of FF1. This signal becomes the $\overline{\text{WAIT}}$ signal for the CPU, and inserts one additional clock period (T_w) into the M1 cycle.

At the onset of clock period T_w, then, FF2 sees a LOW (i.e., the $\overline{\text{WAIT}}$ signal) on its D input. This LOW is transferred to the Q output of FF2. The Q_2 terminal (FF2) is connected to the *set* input of FF1, so this condition forces Q_1 (FF1) HIGH again, thereby terminating the action.

A similar circuit, shown in Fig. 7-8(b), is used to add a wait state to any memory cycle. When the first clock pulse (T1) arrives, the $\overline{\text{MREQ}}$ line goes LOW, forcing the D input of FF1 LOW. At the onset of clock pulse T2, then, this LOW is transferred to the Q output of FF1. At this time Q_1 is HIGH and Q_2 is HIGH, so the output of the NAND gate drops LOW. (Both NAND inputs must be HIGH for the output to be LOW.) This causes the $\overline{\text{WAIT}}$ input of the CPU to become active. But at the onset of T_w, the added clock period, the LOW on Q_1 is transferred to Q_2. This forces one input of the NAND gate HIGH, thereby cancelling the $\overline{\text{WAIT}}$ signal.

MEMORY MAPPED DEVICES

Some peripheral devices used with microcomputers can be more efficiently employed if they are treated as a memory location, instead of an I/O device. An example might be a *digital-to-analog converter* (DAC), which is a device that creates an analog output voltage (or current) that is proportional to a binary digital word applied to its input.

Figure 7-9 shows how an 8-bit DAC can be interfaced with a Z80 *as if the DAC were a memory location.* The DAC requires stable input data, but the data on the bus are transitory. Therefore, we need a *data latch* between the 8-bit data bus and the DAC inputs. There are a number of interface chips that will perform this job, but most of those special-purpose devices are costly. A low-cost solution, which works just as well, is to use a 74100 TTL dual quad-latch. The two four-bit sections of the 74100 become an eight-bit latch when the *strobe* terminals are tied together.

The 74100 latch transfers the information on the data bus to the DAC when the strobe line is HIGH. The 74100 outputs, connected to the DAC

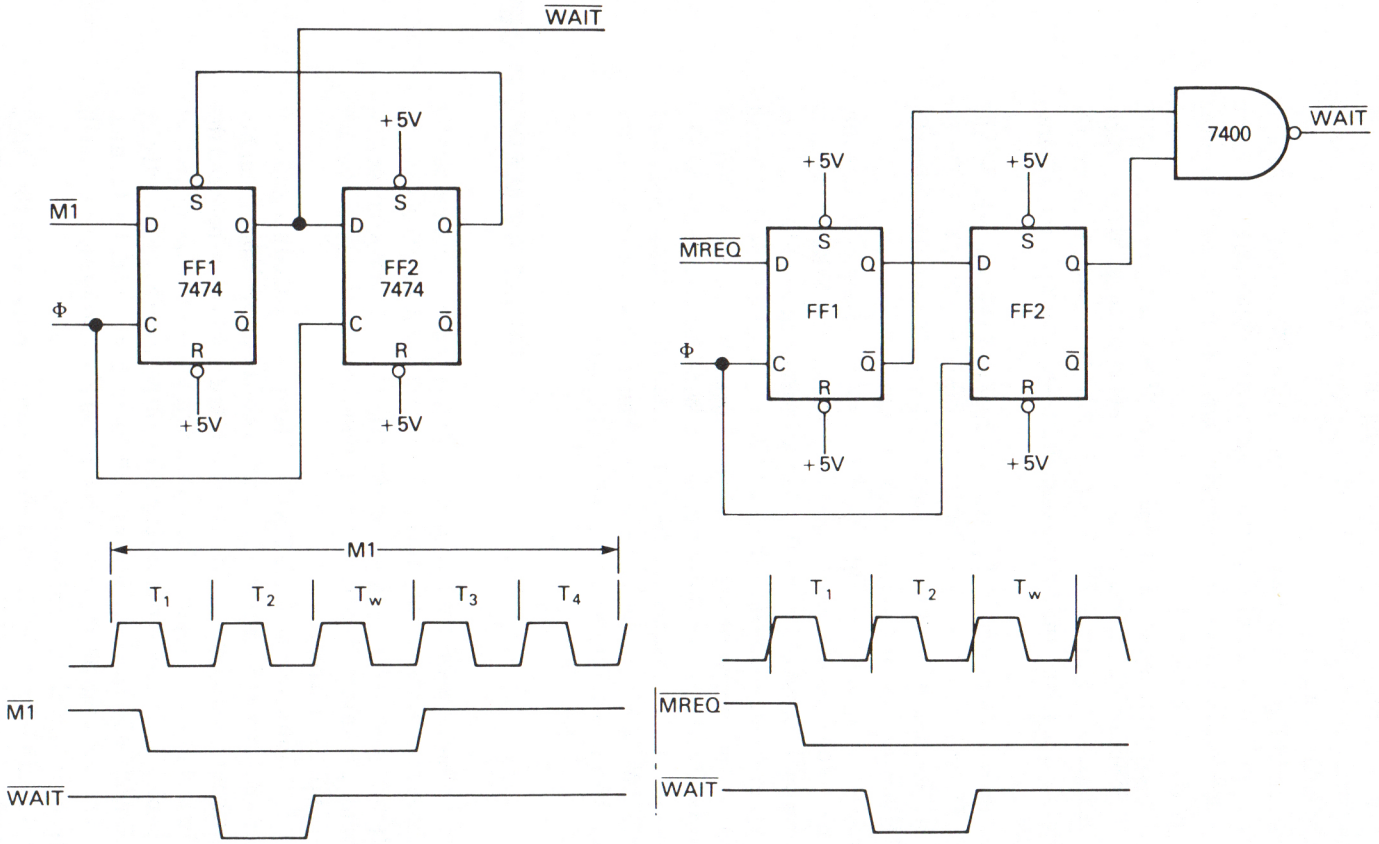


FIGURE 7-8. Adding a wait state to the M1 cycle.

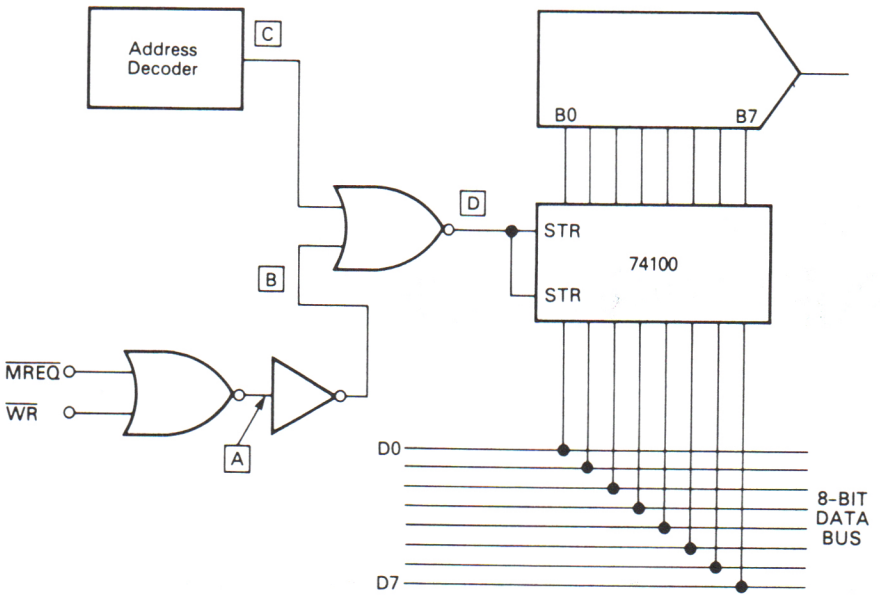


FIGURE 7-9. Using a device such as a DAC as a piece of memory.

inputs, will retain these data when the strobe line again goes LOW. The idea, then, is to make the 74100 strobe line HIGH during the period when the desired DAC input data are present on the data bus.

Three criteria must be met before the data on the bus can be input to the DAC: (1) The write signal ($\overline{WR}$) must be active; (2) the memory request ($\overline{MREQ}$) must be active; (3) the correct address (the address of the location assigned to the DAC) must be present on the address bus. The first two criteria are examined by a single NOR gate. When both $\overline{WR}$ and $\overline{MREQ}$ are LOW (i.e., active), we are producing a memory write operation. This will cause point "A" to go HIGH, and point "B" to go LOW. We do not want the DAC to respond, however, unless point "C" is LOW at the same time. When point "C" is LOW, we know that the address for the DAC is being sent over the address bus.

When all three criteria are met, the strobe input of the 74100 (point "D") will go HIGH. This will allow transfer of data from the data bus into the DAC.

Most microcomputers have less than the full 64K complement of memory. This is the reason why most memory-mapped devices tend to be allocated addresses in the upper 32K of memory. This, incidentally, allows us to use bit A15 of the address bus to discriminate between the various addresses.

CHAPTER 8
INTERRUPT HARDWARE

WHAT IS AN INTERRUPT?

An *interrupt* is a process in which your T/S-1000 stops executing the main program, and begins executing another program located somewhere else in memory. This is not a mere "jump" or "call" operation, but a response to an external stimulus.

There are several reasons why an interrupt capability may be required. One of these is the case of an *alarm* condition. We could, for example, use a computer in an environmental control system, and use the interrupt capability to allow response to alarm situations (e.g., smoke detector, liquid level, burglar alarm, over-temperature, etc.). The computer would ordinarily go about some other chore, perhaps the business of controlling the system. But once during the execution of each instruction of the program, the CPU will interrogate the interrupt system. It is, then, monitoring the alarm status while executing some unrelated program. When an interrupt is received, indicating an alarm status, the computer would jump immediately to the program that *services* the interrupt—rings a bell, calls the fire department, turns on a light, sighs heavily, etc.

Another application is to input data that occurs only occasionally, or whose periodicity is so long as to force the computer to do nothing for an inordinate amount of time. A real-time clock, or timer, for example, might

want to update its input to the computer only once per minute. An analog-to-digital converter (ADC) might have a 20-millisecond conversion time. Even the slower version of the Z80 CPU chip (using a 2.5-mHz clock) can perform hundreds of thousands of operations while waiting for that ADC to complete its conversion job. Since the ADC will not provide valid data until after the conversion time expires, waiting for those data would be a tremendous waste of CPU time.

Another use is to input, or output, data to/from a peripheral device such as a line printer, teletypewriter, keyboard, terminal, etc. Those electromechanical devices are notoriously slow to operate. Even so-called "high-speed" line printers are considerably slower than the Z80 CPU. A classic example is the "standard" 100-word-per-minute teletypewriter. A "word," in this case, is five ASCII characters, so we have to output 500 characters per minute to operate at top speed. This is a rate of 8 characters per second, so each character requires $\frac{1}{8}$ of a second, or 125 milliseconds, to print. The CPU, on the other hand, is a trifle faster. It can output the character to the input buffer of the teletypewriter in something like 3 *microseconds*. The Z80 can execute almost 42,000 outputs in the time it takes the teletypewriter to print just one character.

There are at least two ways to handle this situation, and both involve having the peripheral device signal the CPU when it is ready to accept another character. This is done by using a strobe pulse from the peripheral, issued when it is ready to receive (or deliver) another data byte. One way to handle this problem is to have the programmer write in a periodic poll of the peripheral. The strobe pulse is applied to one bit of an input port. A program is written that periodically examines that bit to see if it is HIGH. If it is found to be HIGH, then the program control will jump to a subroutine that services the peripheral. But this approach is still wasteful of CPU time, and places undue constraints on the programmers's freedom.

A superior method is to use the computer's interrupt capability. The peripheral strobe pulse becomes an *interrupt request*. When the CPU recognizes the interrupt request, it transfers program control to an interrupt service subroutine (i.e., a program that performs some function required for the operation of the peripheral that generates the interrupt). When the service program is completed, then control is transferred back to the main program at the point where it left off. Note that the CPU does not recognize an interrupt request until after it has finished executing the current instruction. Program control then returns to the *next* instruction in the main program that would have been executed had no interrupt occurred.

TYPES OF Z80 INTERRUPT

There are two basic types of interrupt recognized by the Z80 CPU: *nonmaskable* and *maskable*. The nonmaskable interrupt is executed next in

sequence regardless of any other considerations. The maskable interrupts, however, depend upon the condition of an interrupt flip-flop inside of the Z80. If the programmer wishes to mask, i.e., ignore, an interrupt, then the appropriate flip-flop is turned off.

There are three distinct forms of maskable interrupt in the Z80, and these take the designations *mode-Ø, mode-1, and mode-2.*

There are two interrupt input terminals on the Z80 chip. The $\overline{\text{NMI}}$ (pin 17) is for the nonmaskable interrupt, while the $\overline{\text{INT}}$ is for the maskable interrupts.

The nonmaskable interrupt ($\overline{\text{NMI}}$) is much like a *restart* instruction, except that it automatically causes program control to jump to memory location ØØ 66 (hex), instead of to one of the eight standard restart addresses. Location ØØ 66 (hex) must be reserved by the programmer for some instruction in the interrupt service program, very often an unconditional jump to some other location higher in memory.

The mode-Ø maskable interrupt causes the Z80 to pretend that it is an 8080A, preserving some of the software compatibility between the two CPUs. During a mode-Ø interrupt, the interrupting device places any valid instruction on the CPU data bus, and the CPU executes this instruction. The time of execution will be the normal time period for that type of instruction, plus two clock pulses. In most cases, the interrupting device will place a *restart* instruction on the data bus, because all of these are one-byte instructions. The restart instructions transfer program control to one of eight page-Ø locations.

Any time that a $\overline{\text{RESET}}$ pulse is applied (i.e., pin 26 of the Z80 is brought LOW), the CPU automatically goes to the mode-Ø condition. This interrupt mode, like the other two maskable interrupt modes, can be set from software by executing the appropriate instruction (in this case, an IM Ø instruction).

The mode-1 interrupt is selected by execution of an IM1 instruction. Mode-1 is totally under software control, and cannot be accessed by using a hardware action. Once set, the mode-1 interrupt is actuated by bringing the $\overline{\text{INT}}$ line LOW momentarily. In mode-1, the Z80 will execute a restart to location ØØ 38 (hex).

The mode-2 interrupt is, perhaps, the most powerful of the Z80 interrupts. It allows an *indirect* call to any location in memory. The 8080A device (and the Z80 operating in mode-Ø) permits only eight interrupt lines. But in mode-2, the Z80 can respond to as many as 128 different interrupt lines.

Mode-2 interrupts are said to be *vectored*, because they can be made to jump to any location in the 65,536 bytes of memory. In Chapter 16 we will discuss this mode more fully.

INTERRUPT HARDWARE

In this section we will discuss some of the circuitry needed to support the Z80 interrupt capability. Note that the primary emphasis in this chapter will be low-cost circuits not necessarily intended originally for use with the Z80. Keep in mind, however, that Zilog, Mostek, and others manufacture some sophisticated interrupt controller devices, or build into PIO and SIO chips the ability to control interrupts.

INTERRUPT REQUESTS

In the simplest cases, interrupt request lines can be built simply by extending the $\overline{\text{INT}}$ and/or $\overline{\text{NMI}}$ lines to the peripheral device. This assumes a very simple arrangement, in which only one peripheral is to be serviced. Figure 8-1 shows how this might be accomplished. The $\overline{\text{NMI}}$ line (pin 17) is brought out as a monmaskable interrupt line. The optional pull-up resistor (R1) is used to insure that pin 17 remains at the HIGH condition, and thereby helps reduce noise response.

The $\overline{\text{INT}}$ line can be treated in exactly the same manner if there is to be

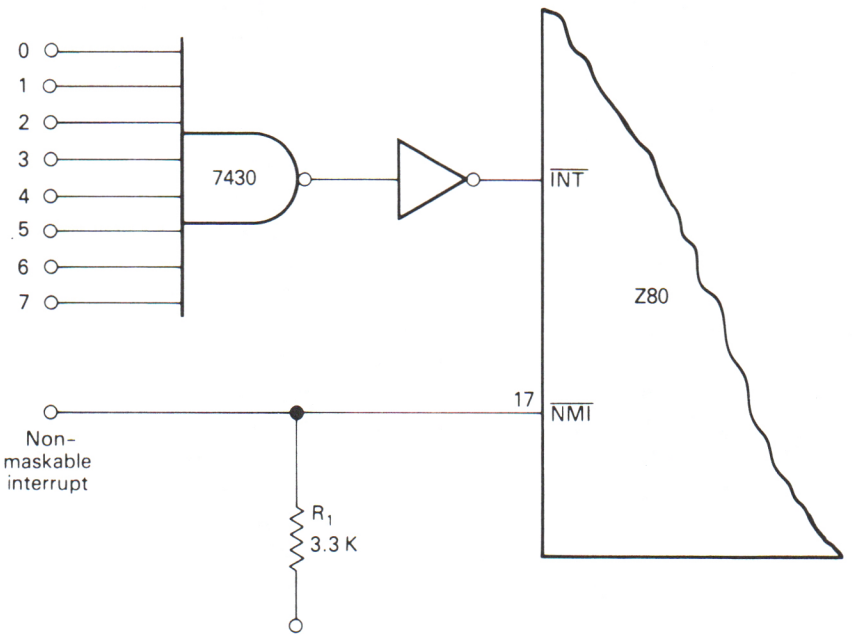


FIGURE 8-1. Interrupt interfacing.

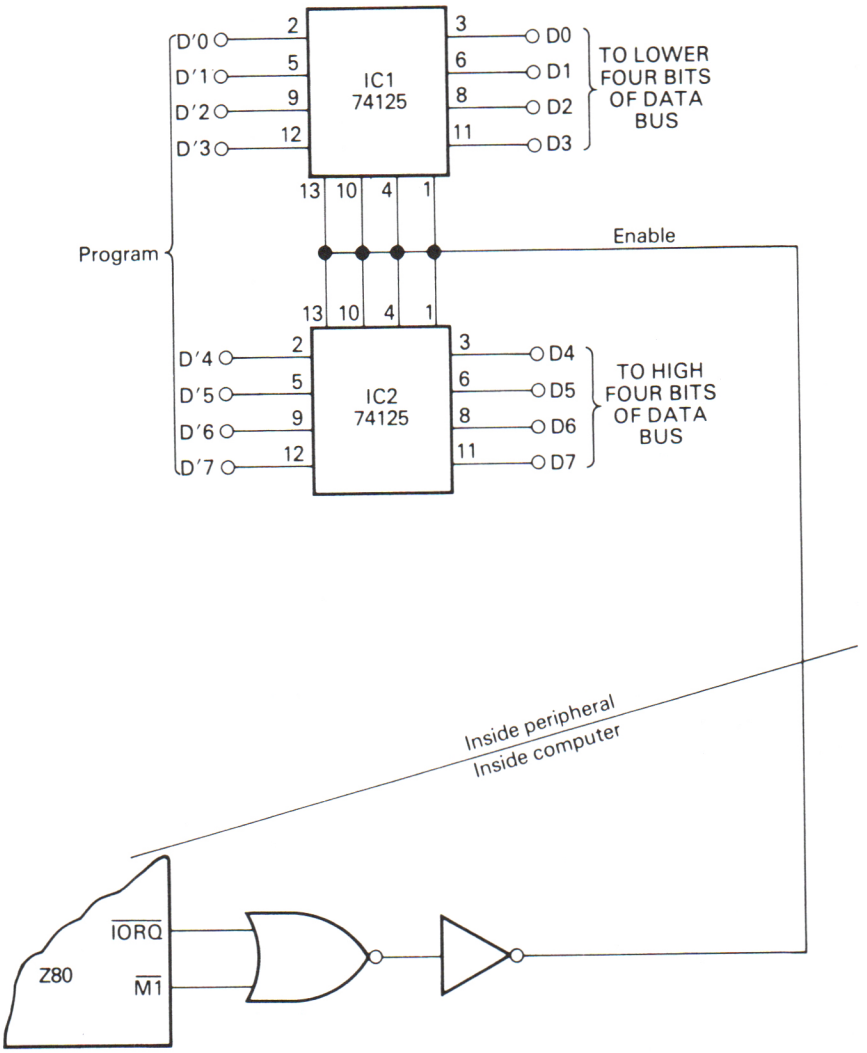
but one interrupting peripheral. But in this case, we have demonstrated how the same pin might be used to recognize up to eight interrupts. This arrangement can be used if only mode-Ø is anticipated. The peripheral that generates the interrupt then places the correct restart instruction on the data bus. The specific restart instruction received tells the CPU which peripheral initiated the interrupt. The key to this $\overline{\text{INT}}$ circuit is the eight-input TTL NAND gate (i.e., a 7430 IC). If any one of its inputs, which form $\overline{\text{INTERRUPT REQUEST}}$ lines, goes LOW, then the 7430 output goes HIGH. This forces the output of the inverter LOW, creating the needed $\overline{\text{INT}}$ signal at pin 16 of the Z80.

INTERRUPT ACKNOWLEDGE

The CPU will always finish executing the current instruction before recognizing an interrupt request. There is, therefore, a slight delay between the initial request and the time when the CPU is ready to process that request. We need some type of signal to tell the peripheral that generated the interrupt request when the CPU is ready to do business. The Z80 samples the $\overline{\text{INT}}$ line on the rising edge of the last clock pulse of the current instruction. If the $\overline{\text{INT}}$ line is LOW, then the CPU responds by generating an $\overline{\text{IORQ}}$ (input/output request) signal during the next M1 machine cycle. We can, then, accept simultaneous existence of LOW conditions on $\overline{\text{IORQ}}$ and $\overline{\text{M1}}$ to form the interrupt acknowledge signal.

Figure 8-2(a) shows an interrupt acknowledge scheme that works for a single interrupt line. We assume that one of the interrupt request schemes of Fig. 8-1 is also used. The 74125 (IC1/IC2) is a quad, tri-state, TTL buffer. Each 74125 contains four noninverting buffer amplifiers that accept TTL inputs, and provide TTL outputs. When a control line is HIGH, the associated buffer output will "float" in the high-impedance "tri-state" mode. But if the control line is brought LOW, the buffer turns on and operates like any other TTL buffer. The control lines for all eight tri-state buffers (four from each 74125) are tied together to form a single *enable* line. The 74125 devices are located inside the peripheral device.

The particular restart instruction designated to service that particular peripheral must be programmed onto the inputs of the 74125s. For example, if we want the peripheral to cause a jump to the RST 10 location (i.e., memory location ØØ Ø1), then we must place D7 (hex), or 11010111 (binary), on the data bus following the acknowledgment of the interrupt request. We program this value by setting the DØ, D1, D2, D4, D6, and D7 inputs of IC1/IC2 to HIGH (binary 1), and the D3 and D5 inputs to LOW (binary Ø). This enable line is connected to the inverted output of the NOR gate that detects the interrupt acknowledge condition (i.e., the simultaneous LOW on $\overline{\text{IORQ}}$ and

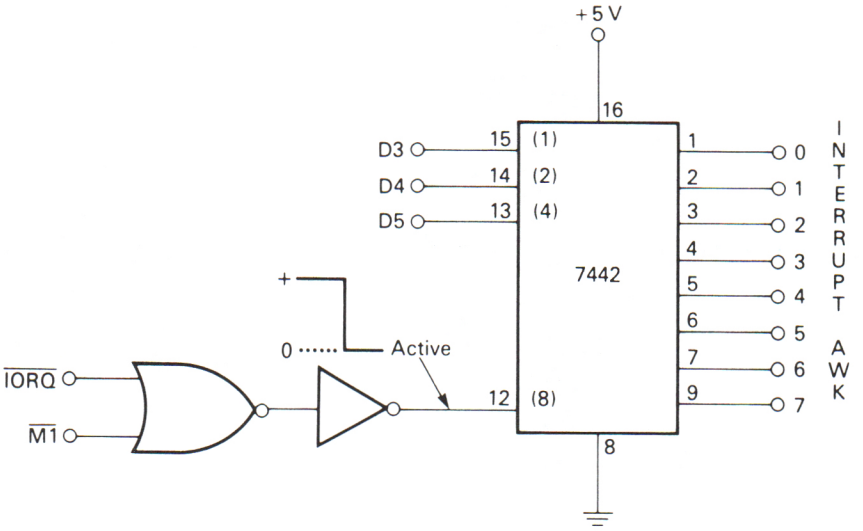


(a)

FIGURE 8-2. (a) Interrupt acknowledge circuit.

$\overline{M1}$). The enable line ordinarily remains HIGH, causing the 74125 outputs to float at high impedance. When the brief interrupt acknowledge pulse comes along, this line momentarily drops LOW, thereby transferring the word (D7 hex) at the 74125 inputs to the data bus. The CPU will decode this instruction, and perform a restart jump to ØØ 1Ø (hex).

Although there is a practical limit to how many tri-state outputs one can easily float across the data bus, we find it quite easy to connect all eight



(b)

FIGURE 8-2 (*Continued*). (b) Interrupt acknowledge for more than one device.

allowed in mode-Ø, and a few more. But how do we differentiate between the peripherals? All will generate the same interrupt request, and these can be handled by using a multi-input NAND gate (see Fig. 8-1 again). But how do we decode the restart instruction given and then send the interrupt acknowledgment to only the *correct* peripheral? Chaos would result if we sent the signal to all eight (or more!) peripherals at the same time. It is very often useful to examine the range of *possible* binary words that are to be used in any given situation. For the mode-Ø interrupt, we are going to use one of eight restart locations, each having its own unique RST op-code. These are listed in Table 8-1. Note that, for all possible states, only three bits change: D3, D4, and D5. The other bits (DØ, D1, D2, D6, and D7) remain constant in all cases (in this particular example, they are all HIGH, but the *important* thing is that they remain at one level in all cases). We can, then, press the 7422 one-of-10 decoder into service once again [see Fig. 8-2(b)]. Recall that the 7422 is a four-bit (BCD) to one-of-10 decoder. The BCD inputs are weighted 1-2-4-8. The 1-2-4 inputs are connected to the D3-D4-D5 lines of the data bus. The 8 line of the 7442 is used as the control line, and is connected to the interrupt acknowledge signal.

In normal, noninterrupt operation, the 8-input of the 7442 is kept HIGH, so the lower eight outputs can never be LOW (when 8 is HIGH, only the 8 and 9 outputs can be active). But when the interrupt acknowledge signal is generated, the 7442 detects the condition of the D3-D5 lines of the data bus, and issues the appropriate signal. The only problem that must be considered

Table 8-1 RST n CODES FOR INTERRUPTS Ø-7

INTERRUPT	RST n	HEXADECIMAL	BINARY
Ø	ØØ	C7	11000111
1	Ø8	CF	11001111
2	1Ø	D7	11010111
3	18	DF	11011111
4	2Ø	E7	11100111
5	28	EF	11101111
6	3Ø	F7	11110111
7	38	FF	11111111

is the possibility that more than one peripheral will attempt to interrupt at one time. This could cause confusion, to say the least. In a moment we will consider methods for prioritizing the interrupts.

Figure 8-3 shows a decoding scheme that can be used inside the computer, and will allow single line selection for up to eight interrupt lines in mode-Ø. We are using 74125 quad, tri-state buffers in the same manner as in Fig. 8-2. But notice in Table 8-1 that the least significant four bits of each restart instruction op-code is always either a 7 or F (both hex). Furthermore, the most significant four bits will be one of four possible states C, D, E, or F. We can, then, create all eight possible op-codes by using only six 74125s and some gates, instead of 16 (as would be required if Fig. 8-2 were implemented for all eight!). The inputs of the 74125s are programmed as follows:

IC1 7
IC2 F
IC3 C
IC4 D
IC5 E
IC6 F

The key to our decoding scheme is to gate on the enable lines of only the appropriate 74125s. IC1 and IC2 form the code for the least significant half-byte of the op-code. There are four interrupt lines that should turn on IC1, and the other four should turn on IC2. We may use a 7420 four-input NAND gate to select which is turned on. If any input of a NAND gate goes LOW, then its output is HIGH. We connect the respective inputs of gate G1 to those interrupt lines that want a "7" in the least significant spot; i.e., Ø, 2, 4, and 6 (see Table 8-1). If any of these four interrput lines goes LOW, then IC1 is turned on, and a hex 7 is output to the lower-order half-byte of the data bus.

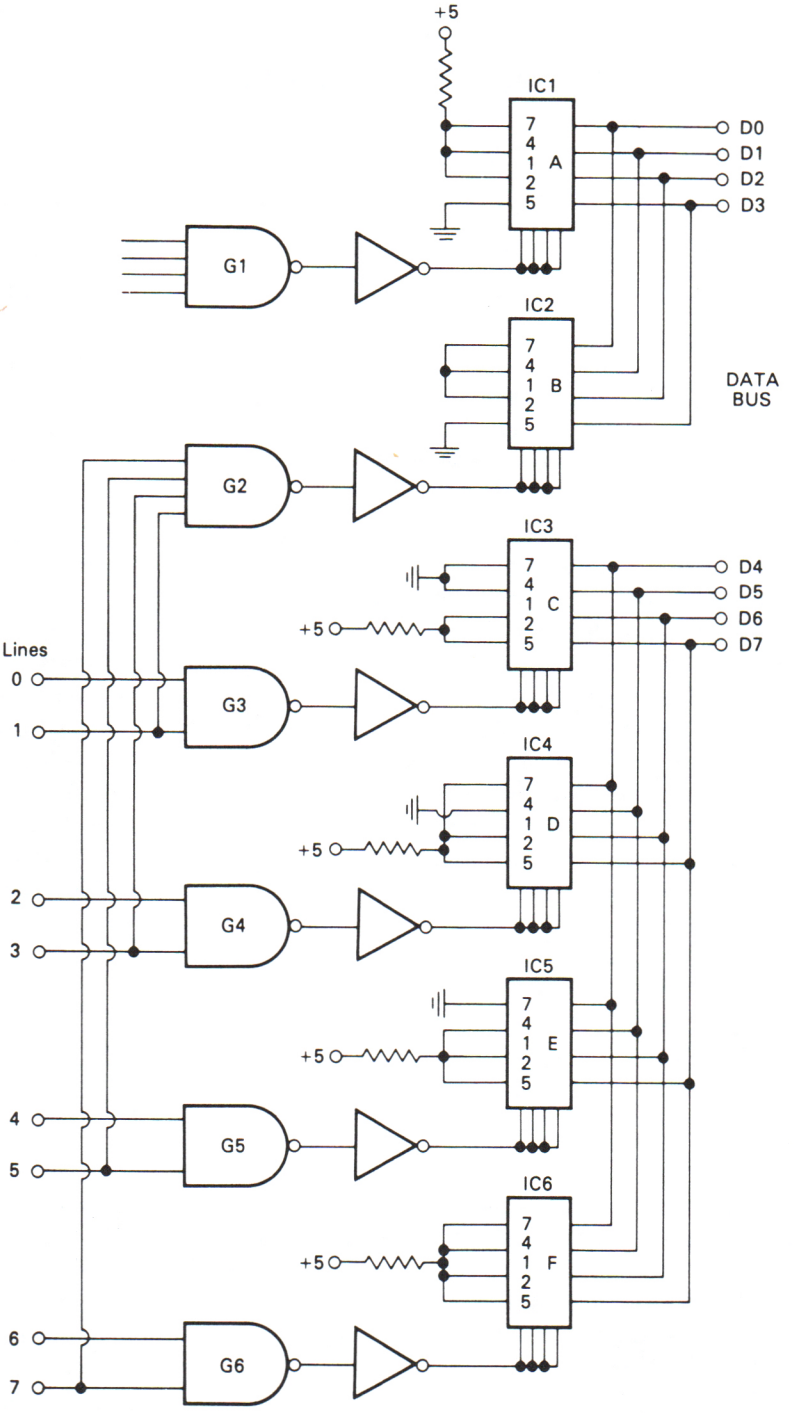


FIGURE 8-3. Generating RST codes for multiple-device interrupts.

Similarly, gate G2 controls IC2. Its inputs are connected to the 1, 3, 5, and 7 interrupt lines. If any of these lines goes LOW, then a hex F is output to the lower-order half-byte of the data bus.

A similar scheme is used to control the higher-order half-byte of the op-code. But in this case, we have four possibilities, each affecting two interrupt lines. IC3-IC6 form the high-order half-byte of the op-code. Since each of these ICs affects only two interrupt lines, gates G3-G6 need only two inputs. These are connected as follows:

GATE	INTERRUPT LINES
G3	Ø,1
G4	2,3
G5	4,5
G6	6,7

If either interrupt Ø or interrupt 1 becomes active, that line will go LOW, causing the output of G3 to go HIGH. This signal is inverted and applied to IC3, which outputs a hex C onto the data bus. Similarly, G4-G6 will cause the appropriate 74125 to output the correct op-code when their interrupts become active.

The schemes above are all relatively simple, and involve the use of ordinary TTL support integrated circuits. But they all also suffer from a common malady. If more than one peripheral device decides to issue an interrupt request, then chaos reigns! The logic to prioritize the interrupt response sequence is much more complex than the circuits shown thus far. Fortunately, there are special purpose integrated circuits, designed for direct interfacing with microprocessor chips, that will allow programming to prioritize and control the interrupts.

In their famous *Bugbook* series, Larsen, Titus, and Rony used a 74148 priority encoder and an Intel 8212 (see *The 8080A Bugbook,* by Rony, et al., Howard W. Sams & Co., Inc., Indianapolis, Indiana 21447) to prioritize interrupts for the 8080A. With little modification the same scheme should work on the Z80 in mode-Ø.

You are also permitted to use the Intel 8214 interrupt controller IC (with a little extra logic), even though it was designed for use with the 8080A. Both 8255 and 8257 devices are also useful in Z80 circuits.

CHAPTER 9
INTERFACING PERIPHERALS

Most microcomputers require additional peripheral devices to make them useful: teletypewriters, printers, video terminals, keyboards, etc. These devices allow the input or output of data, or provide a "hard copy" readout for the end user.

Any number of schemes have been used to connect such devices to the microcomputer, but only a few are described here. Before one goes to a lot of trouble on any one device, it is wise to see if a special-purpose integrated circuit, or a small printed circuit board kit, has been developed to make the interfacing chore a bit easier. New interface chips are being developed all of the time, so check the catalogs of the IC manufacturers. A large number of "basement laboratories" businesses advertise in the hobby computer magazines, and many of these offer kits of parts or printed circuit boards for projects that aid interfacing chores. Most of these are well designed, or are so simple that anybody could do a good job. The prices asked are often small enough to attract even the wisened professional.

STRAIGHT I/O METHODS

Some peripherals can be interfaced to the Z80 directly through an ordinary I/O port. Most such ports (see Chapter 6) offer eight parallel, TTL-

compatible bits. These can be interfaced directly to any other eight-bit, TTL-compatible device.

An example might be certain types of printer. Among several options available at the time that you order the printer is the input format. Some will allow you to select TTL-compatible parallel ASCII input format. The ASCII code is a seven-bit code, so the eighth bit can be used as a strobe. Such a machine can be connected directly to a Z80 I/O port, any of which from Chapter 6 will suffice. It is necessary only that a length of multiconductor cable be provided. The cable must be fitted with the appropriate connectors (usually DB-25 family) so that it can be connected directly.

To make the beast work, you must load the accumulator of the Z80 with the binary code whose bit pattern represents the ASCII character that you wish to print. The contents of the accumulator are then output to the port assigned to the printer. Ordinarily, printers are a lot slower than CPUs, so one has to place a timing loop, or series of no-ops (NOP instruction in Z80 language) between character prints. This scheme will give the printer time to catch up with the speedy CPU.

The standard ASCII code is shown in Table 9-1. From this table, we see that the ASCII code for the character "?" is the binary bit pattern 0111111. If we use the most significant bit (not needed in ASCII) to strobe the printer, then the code will be 10111111, or BF in hexadecimal. To print the question mark ("?") symbol we load the accumulator of the Z80 with BF hex and then output it to the proper port. Suppose, for example, that we wanted to use port 175_{10} for the printer. Such a program might look like:

```
.     .
.     .
.     .
3E    LD A,N

BF    N (i.e., ASCII "?")
D3    OUT 175₁₀,A
AF    175₁₀ in hex
```

Of course, with a "dumb printer," you would have to send it a character, and then waste a lot of time in a loop until you were sure that it has time to print the character and advance a space before sending it another character. But if the machine were a little smarter, it could do some "handshaking." This means that the printer could send a bit back to the CPU to tell it when the next character could be sent. We would loop, or do other chores (interrupts are handy here!), until the strobe is received back from the printer. A typical program might be

Table 9-1 ASCII CODE

	000	001	010	011	100	101	110	111
0000	NULL	① DC_0	b̸	0	@	P	Unassigned	
0001	SOM	DC_1	!	1	A	Q		
0010	EOA	DC_2	"	2	B	R		
0011	EOM	DC_3	#	3	C	S		
0100	EOT	DC_4 (stop)	$	4	D	T		
0101	WRU	ERR	%	5	E	U		
0110	RU	SYNC	&	6	F	V		
0111	BELL	LEM	'	7	G	W		
1000	FE_0	S_0	(	8	H	X		
1001	HT / SK	S_1	)	9	I	Y		
1010	LF	S_2	*	:	J	Z		
1011	V_{TAB}	S_3	+	;	K	[		
1100	FF	S_4	(comma) ,	<	L	\		ACK
1101	CR	S_5	–	=	M	]		②
1110	SO	S_6	.	>	N	↑		ESC
1111	SI	S_7	/	?	O	←		DEL

Example: | 100 | 0001 | = A

b_7 – – – – – – – b_1

The abbreviations used in the figure mean:

NULL	Null Idle	CR	Carriage return
SOM	Start of message	SO	Shift out
EOA	End of address	SI	Shift in
EOM	End of message	DC_0	Device control ① Reserved for data Link escape
EOT	End of transmission	DC_1 - DC_3	Device control
WRU	"Who are you?"	ERR	Error
RU	"Are you . . . . ?"	SYNC	Synchronous idle
BELL	Audible signal	LEM	Logical end of media
FE	Format effector	SO_0 - SO_7	Separator (information)
HT	Horizontal tabulation		Word separator (blank, normally non-printing)
SK	Skip (punched card)	ACK	Acknowledge
LF	Line feed	②	Unassigned control
V/TAB	Vertical tabulation	ESC	Escape
FF	Form feed	DEL	Delete Idle

```
ØØØ1   DB   IN 175₁₀
ØØØ2   AF   175₁₀ in hex
ØØØ3   CB   Test Bit-7 of accumulator
ØØØ4   7F
ØØØ5   20   JR NZ
ØØØ6   86   (Jump−4)
ØØØ7   3E   LD A, N
ØØØ8   BF   N (ASCII for "?")
ØØØ9   D3   OUT 175₁₀
ØØØA   AF   175₁₀ in hex
ØØØB   C3   JP ØØØ1
ØØØC   Ø1   N (L byte)
ØØØD   ØØ   N (H byte)
```

Again, we are using port 175_{10}. The handshaking strobe pulse from the printer is applied to bit-7 (i.e., MSB) of the input port 175_{10}. The output data to the printer are applied to the output side of port 175_{10}. We are assuming that the strobe will go HIGH (i.e., logical-1) when active. Instructions 1–6 are a test and loop program that seeks the condition of the strobe bit. We first input the contents of 175_{10} (instructions 1–2), and then test bit-7 of the input data (instructions 3–4). If this bit is Ø, indicating that the printer is not yet ready, the program counter (PC) is told to pick up the instruction four locations back from JR NZ (i.e., location ØØØ1). These instructions cause the Z80 to continuously input data from the printer, and test to see if the strobe is active. If the strobe is not active, the program loops back to the input, and starts over again.

When the condition is met (i.e., bit-7 is 1), then the program will fall through and output the ASCII character.

Following the output operation, the program control is transferred back to ØØØ1 by a JP NN instruction.

We could also write a similar program to input data from a keyboard. Again, let us assume that the strobe is active HIGH (in real life, do not *assume;* examine that strobe. Many are active LOW, especially on keyboard units), and that it is applied to the MSB of port 175_{10}. We will continue to loop until a HIGH is received at bit-7 of port 175_{10}:

```
ØØØ1   DB   IN 175₁₀
ØØØ2   AF   175₁₀ in hex
ØØØ3   CB   Test bit-7
ØØØ4   7F
```

```
ØØØ5   20   JR NZ
ØØØ6   86   (Jump−4)
ØØØ7   ·    (program to do something with
ØØxx   ·    input data)
ØØxx   ·
ØØxx   C3   JP ØØØ1
ØØxx   Ø1   N (L byte)
ØØxx   ØØ   N (H byte)
```

As in the previous example, the CPU inputs from port 175_{10} and tests bit-7. If bit-7 is LOW, program control jumps back to ØØØ1, and the loop is repeated, But if bit-7 is HIGH, the program continues on to location ØØØ7. The instruction in this location must *do something* with the data input from port 175_{10} in instruction step ØØØ1. Since it is a keyboard routine, we usually PUSH A onto an external memory stack.

Following the portion of the program that "does something" with the ASCII character input in ØØØ1, we jump immediately back to ØØØ1 to repeat the routine.

TELETYPEWRITERS

Teletypewriters have been around for several decades, and form one of the lowest-cost hard-copy devices used in computer work. Most machines are manufactured by the Teletype Corporation, of Skokie, Illinois. The word *Teletype* is a registered trademark of that company, while the word *teletypewriter* is a generic term.

Most of the older model Teletypes[R] (i.e., models 15, 19, 28, etc.) are encoded in the Baudot code system, a five-bit code. The newest models (33 and 43) are encoded in the ASCII code system (see Table 9-1).

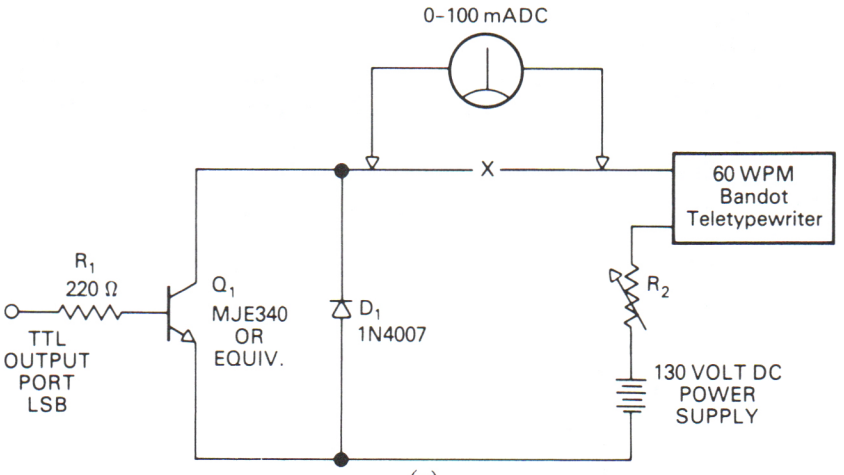
The popularity of teletypewriters is due to their low initial cost and relatively good reliability. For hobbyists, they are popular because they are available at relatively low cost on the government and commercial surplus market. For most hobby applications, and for many commercial, scientific, or engineering applications, the Model 33 Teletype[R] is a best bet (around $500 used and in good condition). This machine is encoded in ASCII, so it is compatible with most microcomputers, and is easy to maintain.

Like most electric typewriters, teletypewriters use electrical solenoids to pull in the selector arms that actually determine which character is printed. For ASCII (a 7-bit code), seven selector magnets and seven solenoids are used.

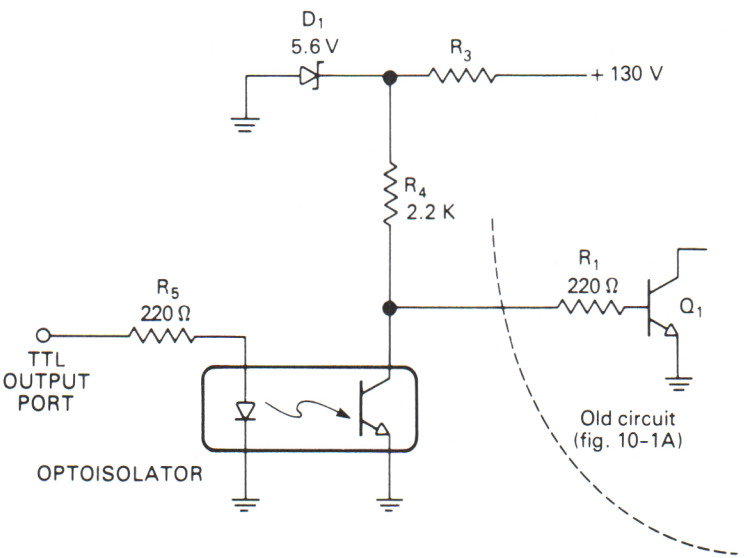
The basis for selection is to pull in those selectors (each selector

represents one bit) common to the desired character. This is done by series-connecting the solenoids together in a circuit called a *current loop*. Older machines used a 60-milliampere current loop, while the 33 and 43 use 20-milliampere current loops.

Figure 9-1 shows the circuit for interfacing a 60-mA machine to a



(a)



(b)

FIGURE 9-1. (a) Simple circuit to interface old-style Baudot teletypewriters. Adjust R2 for 60 mA in the loop; (b) Circuit above modified to isolate the teletypewriter from the computer output circuitry.

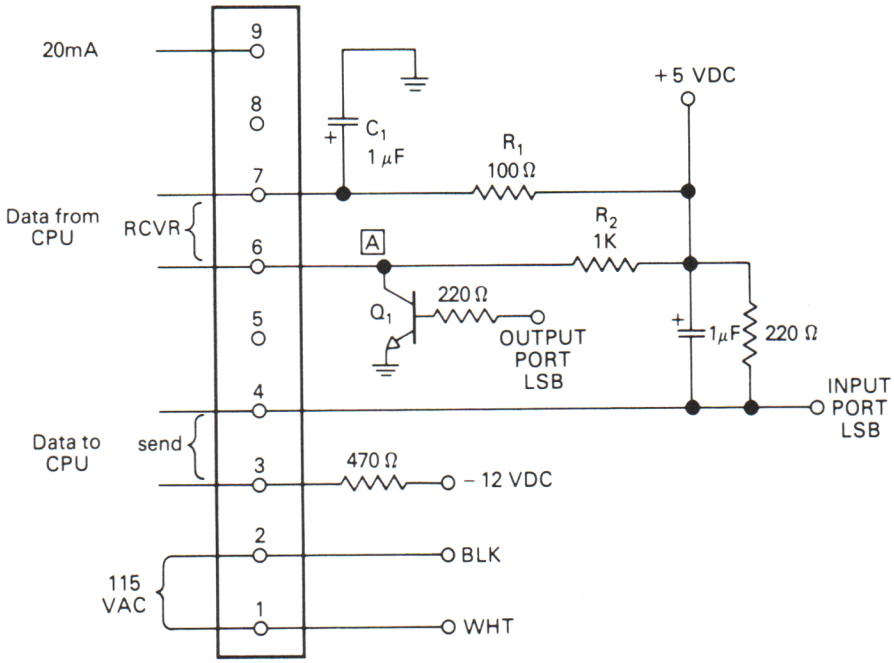
microcomputer, while Fig. 9-2 shows the circuit for a 20-mA machine. In both cases, we use one bit of a TTL-compatible output port (see Chapter 6) for drive to the teletypewriter.

Figure 9-1(a) shows the most basic circuit for a 60-mA machine. An external 130-volt DC power supply is needed. The current loop circuit consists of the DC supply, resistor R2, the teletypewriter machine, and *c-e* path of transistor Q1.

Diode D1 is used as a spike suppressor. The solenoid coils will produce a spike-like pulse (i.e., high amplitude, short duration) every time the current flow in one of the coils is interrupted. Diode D1 is connected to suppress these spikes, and is used mainly to protect transistor Q1.

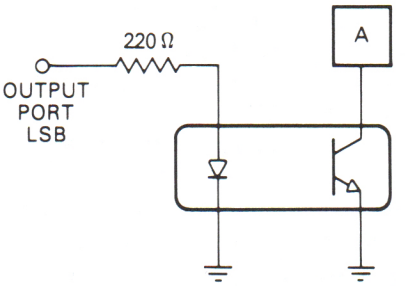
Transistor Q1 can be any high-voltage power transistor that is capable of handling a 60-mA collector current. Q1 acts as a switch to turn the loop on and off.

If a HIGH appears on the LSB of the selected output port, then Q1 is forward-biased. Its *c-e* path conducts current, closing the loop. When the

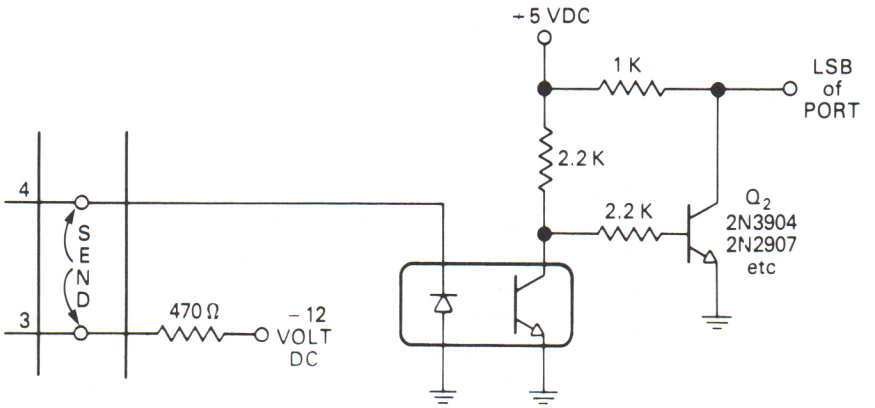


(a)

FIGURE 9-2. (a) Circuit to connect computer output port to the Model 33 teletypewriter. Terminal block shown is found under the top cover of the Mdl. 33, on the right rear when viewed from the front of the keyboard. Use separate +5-volt DC power supplies for best results.



(b)



(c)

FIGURE 9-2 (*Continued*). (b) Modification of the standard circuit to allow isolation of the computer from the teletypewriter. (c) Different circuit to accomplish the same job.

LSB of the output port is LOW, then Q1 is reverse-biased. Under this condition, its *c-e* path is turned off, so the loop is open.

It is best to adjust resistor R2 to obtain a loop current of 60 mA. Place a HIGH on the LSB of the selected port, and press one of the teletypewriter keys. A millammeter placed at the point indicated in Fig. 9-1(a) will show the current. Adjust the resistor (R2) for a flow of 60 mA.

It is probably best if all high-voltage circuits are isolated from your computer's output. A fault in transistor Q1 could otherwise cause damage to the output port circuits. An appropriate circuit for this is shown in Fig. 9-1(b). The secret is to use an optoisolator device. On the computer side of the device is an LED, while on the teletypewriter side is an optotransistor. The transistor will be turned off unless the LED is turned on. The collector of the optoisolator transistor is connected to the point in the previous circuit that connected to the computer. This collector is also connected to a 5.6-volt DC power supply that is derived from the +130-volt DC power supply used in the

current loop. On the computer side, the LED is connected, through a current-limiting resistor (R5), to the LSB of the selected port.

When the LSB of the output port is HIGH, then the LED is turned on. This turns on the transistor in the optoisolator, shorting out the bias to the current loop transistor. This action turns off the loop. Similarly, the LOW on the LSB of the port turns off the transistor, so Q1 is turned on, closing the loop. The action in this circuit is inverted, so it is necessary to complement the Z80 accumulator before outputting data. Alternatively, you could use one other transistor inverter, between the isolator and Q1, to invert the output of the isolator.

Figure 9-2 shows a circuit that is used to interface the model 33 to a Z80 output port. Looking from the front panel, there is a terminal strip on the right-rear side of the Model 33. This terminal strip, shown schematically in Fig. 9-2(a), contains the send/receive connections for the teletypewriter.

The *receive* side of the machine (terminals 6 and 7) contains the loop, so that the solenoids can be pulled in. The *send* side is merely a set of contact closures. In my own experience, this circuit has produced some problems. If the loop is turned on after the microcomputer is loaded and ready to work, a random pulse seems to change a few (important) bits in a few memory locations. The problem is partially relieved by using +5-volt and −12-volt power supplies that are completely divorced from the computer power supply. But I like the approach shown in Figs. 9-2(b) and 9-2(c). We would use R1, R2, and C1 [from Fig. 9-2(a)], but replace Q1 with the transistor from the optoisolator (connect the collector to point "A"). The LED is connected, again through a current-limiting resistor, to the LSB of the selected output port.

We can use the −12-volt supply to drive the LED, or the +5-volt supply (in which case, the polarity is reversed). The isolator transistor (Q1) drives an inverter stage (Q2). When the LED is turned on, Q2 is turned off, so the LSB of the selected input port is HIGH. But if the LED is off, then Q2 is turned on, dropping the LSB of the input port to zero.

Quite a number of small companies that advertise in the hobby computer magazines offer optoisolator, 20-mA current loop kits as add-ons for many computers. Take advantage of them; they are low in cost.

RS-232 INTERFACING

The Electronic Industries Association (EIA) standard RS-232 pertains to a standarized serial data transmission scheme. The idea is to use the same connector (i.e., the DB-25 family), wired in the same manner all of the time, and to use the same voltage levels. Supposedly, one could connect together any two devices that provide RS-232 I/O without any problem (it usually works). Modems, CRT terminals, printers (i.e., Model 43 Teletypewriter),

and other devices will be fitted with RS-232 connectors. Some computers provide RS-232 I/O, and this feature can be added by using a set of Motorola ICs called RS-232 drivers/receivers. An RS-232 driver IC accepts TTL outputs from a computer or other device, and produces RS-232 voltage levels at its output. The RS-232 receiver does just the opposite. It takes RS-232 levels from the communications/interface and produces TTL outputs.

Unfortunately, the RS-232 is a very old standard, and it predates even the TTL standard. That is why it uses such odd voltage levels for logical-1 and logical-Ø.

Besides voltage levels, the standard also fixes the load impedances and the output impedances of the drivers.

There are actually two RS-232 standards—the older RS-232B and the current RS-232C (see Fig. 9-3). In the older version, RS-232B, logical-1 is any potential in the −5- to −25-volt range, and logical-Ø is anything between +5 and +25 volts. The voltages in the range −3 to +3 are a transition state, while +3 to +5 and −3 to −5 are undefined.

The speedier RS-232C standard narrows the limits to ±15 volts. In addition, the standard fixes the load resistance to the range 3000 to 7000 ohms, and the driver output impedance that is low. The driver must provide a slew rate of 30 volts/microsecond. The Motorola MC1488 driver and MC1489 receiver ICs meet these specifications.

The standard wiring for the 25-pin DB-25 connector used in RS-232 ports is shown in Table 9-2.

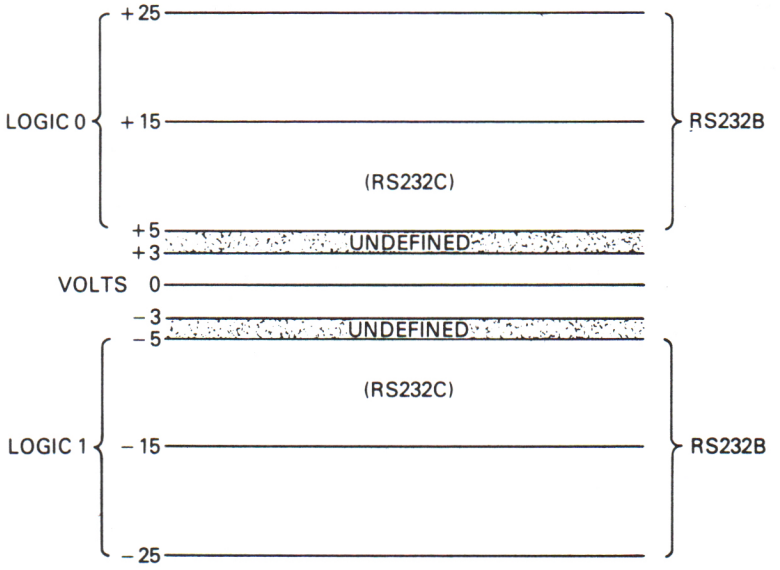


FIGURE 9-3. Diagram showing meaning of the RS-232 EIA standards.

Table 9-2 EIA RS-232 PIN-OUTS FOR STANDARD DB-25 CONNECTOR

PIN NO.	RS-232 NAME	FUNCTION
1	AA	Chassis ground
2	BA	Data from terminal
3	BB	Data received from modem
4	CA	Request to send
5	CB	Clear to send
6	CC	Data set ready
7	AB	Signal ground
8	CF	Carrier detection
9	undef	
10	undef	
11	undef	
12	undef	
13	undef	
14	undef	
15	DB	Transmitted bit clock, internal
16	undef	
17	DD	Received bit clock
18	undef	
19	undef	
20	CD	Data terminal ready
21	undef	
22	CE	Ring indicator
23	undef	
24	DA	Transmitted bit clock, external
25	undef	

CHAPTER 10

EXTERNAL POWER SUPPLIES AND "GLITCH WITCHES"

The little power supply module which comes with the Timex T/S-1000 is quite sufficient to power the computer and its add-on 16K memory module. Some other vendors also use it to supply modest amounts of power to their own add-on accessories. But the truth is that the power supply is limited to 650 milliamperes, and that is not much when you begin hanging multiple TTL and display devices onto the interface connector, and expect them to work nicely. For this type of application, we might well have to supply an external DC power supply. In this chapter we will deal with the power supply that you might want to use for interfacing chores.

Another topic which we will deal with in this chapter—because its cures are mostly in the DC power supply—is the little business of the "Glitch Witch." A "glitch" is an occasional electrical anomaly that gets into your computer and causes it to do dumb things . . . sort of like *R2D2* when his big golden buddy bops him on the . . . . *err* . . . . head. The "witch" part of the name comes from the fact that the causes of these problems seem somehow magical and occult, but none the less real. The Glitch Witch occurs from power line (AC mains) transient conditions that raise hob with your computer. They are the main cause of many seemingly unfounded "bomb-outs" in the computer program execution.

DC POWER SUPPLIES

Most digital IC devices which we will use in interfacing are designed to operate from a +5 volt DC regulated power supply. The power supply limits are usually advertised as +4.75 to +5.2 volts for TTL, but I personally feel this range is a little generous. On the low end of the scale, there are many complex function devices that seem to operate in an erratic manner when the voltage is close to the lower limit of +4.75 volts DC. These devices seem to perform best only when the DC power supply voltage is +4.9 volts or higher. On the high end of the scale, the TTL devices tend to overheat somewhat when powered from a +5.2 volt DC source. For this reason, I personally prefer to reduce the operating limits from the published values to +4.90 to +5.05 volts DC.

The voltage used by the digital devices must be regulated so that it stays within the limits. In this chapter, we will discuss how to obtain such a power supply.

Most texts on microcomputer/microprocessor interfacing assume that the reader has available some form of microcomputer/trainer on which to try out the methods taught either as part of a classroom situation for students, or, on the bench for a working journeyman engineer. It is usually assumed in such texts that the student/reader's microcomputer is ready-made and of the "plug & chug" variety. Unfortunately, many microcomputer trainers come from the manufacturer less power supply! There are many machines on the market that lack a power supply and you might not read the fine print in the advertisement until it is too late. In other cases, you might have an extensive collection of "used" or "scrap" components on which to base a microcomputer power supply. In still other cases, you might be using either a custom designed or "standard" single-board computer such as the *Pro-Log* series and may well want to design your own power supply for the beastie. A shortcoming of some books in this field is the lack of instructions on what to do for a power supply! Let's *rectify* the problem (bad pun, *I know!*).

The circuit for a basic power supply is shown in Fig. 10-1. There is nothing special in the circuit of Fig. 10-1, it is based on well known principles and easily available components.

The heart of any DC power supply is the transformer. The purpose of the transformer in the power supply is to adjust the voltage from the AC power mains down (or up!) to the voltage levels required by electronic circuits. In the case of the typical small single-board trainer computer, the required voltages are +5 volts at a current rating of 3 to 6 amperes, and ±12 volts DC at a current rating of 1 ampere for each potential (total of 2 amperes). The 5 volt potential is obtained from a 6.3 volt AC filament transformer of appropriate rating. The extra voltage (which rectifies to +8 volts DC on peaks) is needed because the regulator input voltage must be 2.5

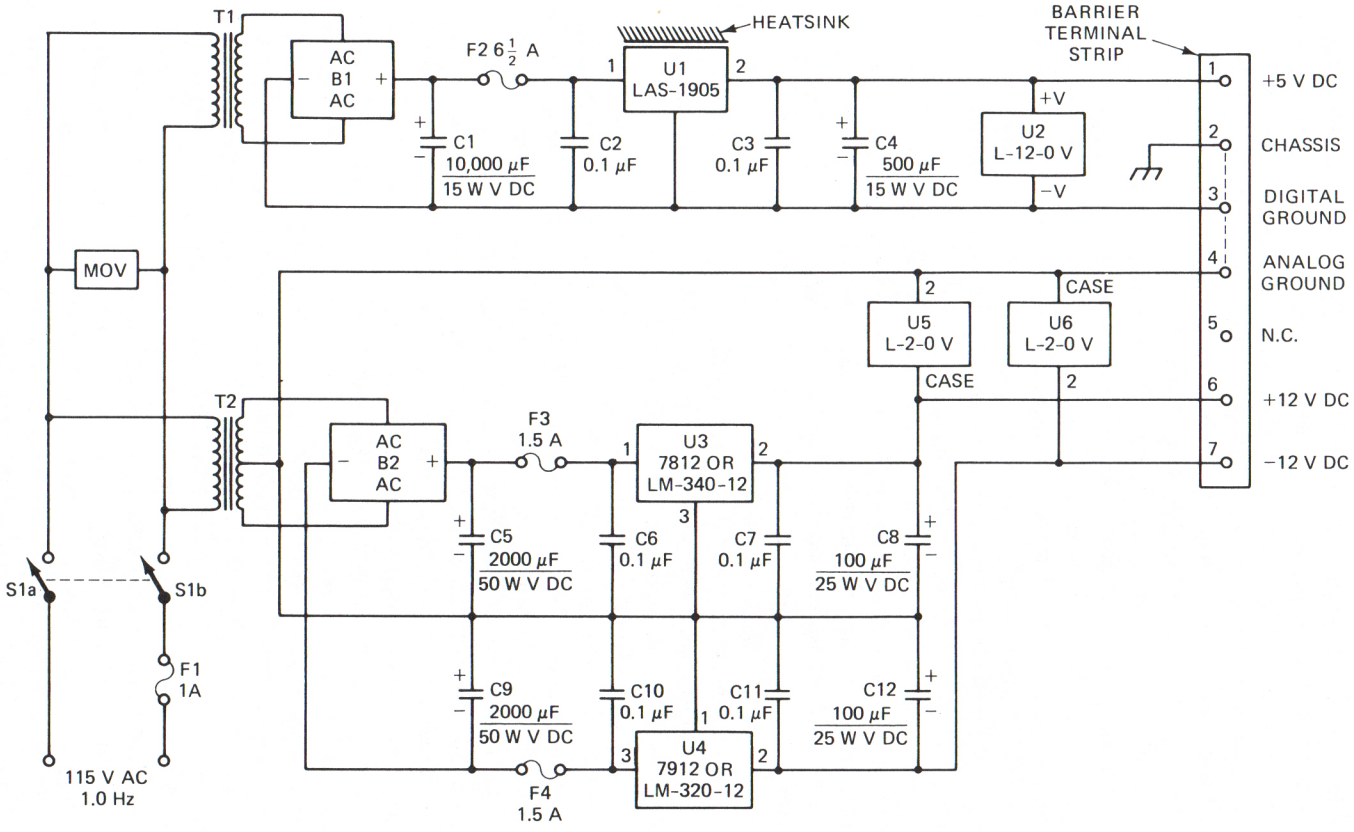


FIGURE 10-1. General purpose power supply circuit.

volts, or more, greater than the rated output voltage. For a 5 volt regulator, therefore, we would require at least 7.5 volts DC into the voltage regulator. A suitable transformer for this project is the *Triad* type F24U which is rated at 6.3/7.5 volts RMS at 8 amperes of secondary current. The "6.3/7.5" designation for the secondary voltage comes from the fact that the transformer has a tapped primary which will cause the secondary potential to be either 6.3 VAC RMS or 7.5 VAC RMS, depending upon which tap is connected to the hot side of the 115 volt AC power mains. The current rating for this transformer technically should be 10 amperes for a 5 ampere power supply in order to not exceed the primary VA rating. This rating is needed because the transformer is used in the bridge rectified configuration shown in Fig. 10-1. The power supply has run at 5 amperes for many months, however, and no problems have reared their heads. But be aware that, technically, running the full 5 amperes is "pushing it."

The transformer for the ±12 volts DC supply can be any type with a secondary rating of 25 to 28 volts AC RMS (25.6 volts is common) with a current rating of 2 amperes or more (there are several 2.8 ampere models on the market). The absolute necessity, however, is a center-tap on the secondary winding. There are only a few models that have the required center tap. Without the C.T., the circuit will not work.

The rectifiers are standard bridge rectifier stacks. The rectifier for the ±12 volt supply is any 50 PIV @ 1 ampere model. Most builders will use one of the printed circuit models that has radial leads.

The rectifier for the +5 volt supply, however, is a little heavier and must be rated at 5 amperes or more. The PIV rating can be anything over 25 volts. I recommend the use of a chassis-mounted 25 ampere bridge if you obtain it from a blister pack source, and a 12 ampere model if from an industrial supplier. The reason for the difference in apparent rating for what seems to be the same rectifier is that some blister pack distributers seem to overrate their products. This is not so much a reflection of dishonesty as a realization that amateur and hobbyist applications are intermittant in nature.

The three different power supplies in Fig. 10-1 are essentially the same, so only one description will be needed for all three. The filter capacitors are selected according to a 1000 uF per ampere rule for the 12 volt supplies and 2000 uF per ampere for the +5 volt supply. The difference is due to the large dynamic current changes that can be expected from digital circuits. The 0.1 uF capacitors (C2, C3, C6, C7, C10 and C11) at the inputs and outputs of the voltage regulators are meant to provide bypassing for noise transients. In order to be effective, these capacitors must be mounted as close as possible to the body of the regulators. Most builders prefer to mount these capacitors *on* the body of the regulators, despite the fact that this method may be aesthetically less appealing than some other methods. The output capacitors (C4, C8 and C12) are selected according to a 100 uF per ampere rule and are

intended to improve the transient response of the circuit. In the case of a large, short duration transient current demand the excess current can be taken from the capacitor while the regulator catches up.

The voltage regulators used in these projects are three-terminal IC regulators. The 12 volt power supplies use either 7812 or LM-340-12 (either T or K styles) for the +12 volt supply, and either 7912 or LM-320-12 (either T or K) for the −12 volt DC supply. Note the difference in pinouts for these two regulators! Failure to recognize this can be fatal for the regulator!

The 5-ampere regulator for the +5 volt DC supply is a special type in a TO-3 transistor case and is made by Lambda Electronics (515 Broad Hollow Road, Melville, NY, 11747). In this application I recommend the use of a finned heat sink for the regulator—it will cause it to run a lot cooler, which means potentially longer and more reliable operation of the device (as is true of any semiconductor power device).

Overvoltage protection is an absolute necessity for any DC power supply used with computers. The regulator input voltage is higher than the rated output voltage by at least 2.5 volts. If something happens to the regulator (and it often happens), then the higher voltage at the input may appear at the output and *will* (not may) damage the circuits of the computer. TTL electronic devices, after all, can withstand only potentials less than +5.25 volts! There are several methods for supplying protection against the possibility of overvoltage damage, but in this case we are using two more Lambda products, the L-12-OV and L-2-OV overvoltage protection modules. The 2-ampere L-2-OV device must be operated reverse polarity for the −12 volt DC supply (note the diagram). On this device, incidentally, the case is the same as the TO-66 diamond-shaped power transistor case. The V− is the case, while the V+ is pin no. 2 (which would correspond to the emitter on a transistor).

The MOV device across the primaries of the transformers is used to suppress high voltage transients appearing on the AC power line. These transients occur frequently, especially in industrial environments. They will cause the computer to run amuck if they hit at the wrong time.

There are three "grounds" in this circuit: chassis, digital and analog. In most cases, these grounds will be joined together at the power supply, so terminals 2, 3 and 4 will be shorted together. In other cases, separate grounds must be maintained in order to prevent ground loops.

The MOV device mentioned above is used to suppress the Glitch Witch. It will truncate those high voltage spikes which come along the power line with disturbing frequency. The device which I have used is the General Electric, although there are other similar devices on the market. Panasonic, for example, markets a Zenister device which performs this same function.

We can also buy AC outlets which have these types of devices built in.

You will find them advertised in most of the computer magazines, under the rubric of "cleaning up the power line."

Another alternative, although not too attractive to owners of low-cost computers, is to buy a special line isolation transformer, such as those made by Topaz, Sola and others. These transformers are especially designed for suppressing high voltage, short duration transients on AC power lines. I have used these transformers to very good effect on several different brands of computers. The problem for the T/S-1000 user, however, is the cost of the transformer will most likely exceed the cost of the computer and its 16K memory module.

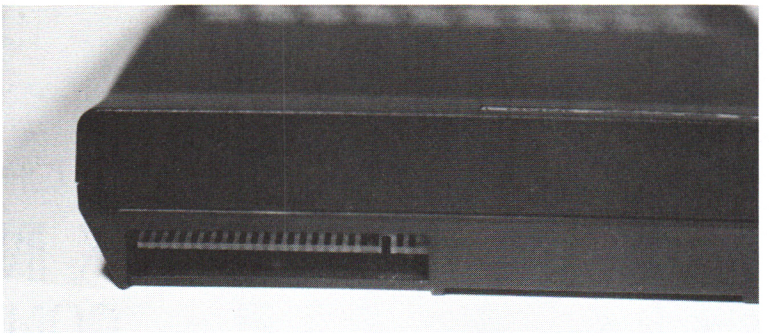
CHAPTER 11
TIMEX INTERFACE CONNECTORS

The art of interfacing microcomputers with the rest of the world has been called the art of replacing hardware and circuits with software. The computer can perform as a "universal analytic instrument," but if it lacks a means for taking data in from an environment other than the keyboard, or if it lacks the ability to interact with external output devices, then there is little that can be done with it other than straight programming (whatever *that* is!). The interface connectors on the Timex computers are what give it versatility.

There are three different Timex computers. Two of these machines, the T/S-1000 and the T/S-1500, have the same interface connector on the rear apron. The T/S-2068 contains an interface connector that incorporates the smaller interface connector of the 1000/1500, but adds certain other signal lines for even greater versatility.

In addition to an expanded interface connector, the T/S-2068 microcomputer also has three other interface connectors of two different types. On either side (i.e. the left and right aprons) there is a 9-pin *Atari*-style game controller connector, while on the front there is another printed circuit board connector which is intended for accepting the external Read Only Memory (ROM) Command Cartridges.

In this chapter we will examine these connectors and define the uses of each pin. Most of the pins contain one of the standard Z80A signals (e.g. ad-



(a)

(b)

FIGURE 11-1 (a) Interface connector on T/S-1000. (b) Diagram (top view) showing connections.

dress bus lines, data bus lines, IORQ, and so forth), which are defined in detail elsewhere in this book.

TIMEX T/S-1000 AND T/S-1500 INTERFACE CONNECTOR

The Timex/Sinclair T/S-1000 microcomputer is equipped with an interface connector on the rear apron of its cabinet (see Fig. 11-1(a)) that is normally used for mounting the 16K add-on memory module. This connector is also usable for adding other accessories, such as 64K memory modules, peripheral modules, I/O ports and so forth. Some manufacturers of T/S-1000 accessories design their products to permit stack-on addition. Each item will have the male printed circuit card-edge connector that mates with the memory module (or other accessories) and the female printed connector that mates with the T/S-1000 male connector.

I personally found the diagram of the interface connector in the T/S-1000 manual a little confusing; it seems to be a mirror image of the actual connector. If you look on page 124 of the manual, the connector pattern shows the slot on the left-hand side of the connector, while it is actually on the right-hand side when the connector is viewed from the rear of the T/S-1000—which is the way most readers will view it! Figure 11-1(b) shows a top view of this connector, while the text below defines the pin-outs.

There are actually two ranges of pins on this connector. The connector is part of the printed foil pattern circuit board, and the pins are spaced on 0.1

inch centers. (Note: many other boards, perhaps the majority, are on 0.125 or 0.156 inch centers, so many connectors are not compatible with the T/S-1000). The 23 pins on the top side of the board are labelled 1A through 23A, while those on the bottom side of the board are labelled 1B through 23B. The respective numbers are opposite each other on the board; that is, 1A is immediately on top of 1B and so forth.

PIN-OUT DESIGNATIONS

Most of the pin-outs on the T/S-1000 interface connector are labelled according to the Z80 function that they serve. The precise definitions of these functions have been discussed elsewhere in this book. Other pins are specific to this computer, but are none the less important when designing interface peripherals.

TOP-SIDE (1A–23A)

1	D7′	Data Bus bit-7
2	$\overline{\text{RAM}}$/ $\overline{\text{CS}}$	Random Access (read/write) Memory chip-select (active-LOW)
3	slot	
4	DØ′	Data Bus bit-Ø
5	D1′	Data Bus bit-1
6	D2′	Data Bus bit-2
7	D6′	Data Bus bit-6
8	D5′	Data Bus bit-5
9	D3′	Data Bus bit-3
10	D4′	Data Bus bit-4
11	$\overline{\text{INT}}$	Interrupt Request (active-LOW)
12	$\overline{\text{NAY}}$	Nonmaskable Interrupt (active-LOW)
13	$\overline{\text{HALT}}$	CPU halt (active-LOW)
14	$\overline{\text{MREQ}}$	Memory Request Line (active-LOW)
15	$\overline{\text{IORQ}}$	Input/Output Request (active-LOW)
16	$\overline{\text{RD}}$	Read signal (active-LOW)
17	$\overline{\text{WR}}$	Write signal (active-LOW)
18	$\overline{\text{BUSAK}}$	Bus Acknowledgment (active-LOW)
19	$\overline{\text{WAIT}}$	Wait signal (active-LOW)
20	$\overline{\text{BUSRQ}}$	Bus Request Signal (active-LOW)
21	$\overline{\text{RESET}}$	Reset signal (active-LOW)
22	$\overline{\text{M1}}$	M1 machine-cycle signal (active-LOW)
23	$\overline{\text{REFSH}}$	Refresh signal (active-LOW)

BOTTOM-SIDE (1B–23B)

1	5V	5 Volts DC regulated (see text)
2	9V	9 Volts DC unregulated (see text)
3	slot	
4,5	ØV	ground connection (two pins are used)
6	$\overline{Ø}$	complement of the clock signal
7	AØ	Address Bus bit-Ø (LSB)
8	A1	Address Bus bit-1
9	A2	Address Bus bit-2
10	A3	Address Bus bit-3
11	A15	Address Bus bit-15 (MSB)
12	A14	Address Bus bit-14
13	A13	Address Bus bit-13
14	A12	Address Bus bit-12
15	A11	Address Bus bit-11
16	A1Ø	Address Bus bit-10
17	A9	Address Bus bit-9
18	A8	Address Bus bit-8
19	A7	Address Bus bit-7
20	A6	Address Bus bit-6
21	A5	Address Bus bit-5
22	A4	Address Bus bit-4
23	$\overline{ROM}$/ $\overline{CS}$	Read Only Memory Chip-Select (active-LOW)

The two DC power connectors can be used to supply a limited amount of DC power to external devices. The current capability, however, is quite limited, so an external power supply will be needed wherever the add-on accessory requires substantial amounts of current. For most projects, a simple +5 volt DC regulated (i.e., TTL) power supply with a current capability sufficient for the project will do nicely. Such power supplies are easily available from mail-order and retail sources, or, they can be built from scratch (see *Elements of Microcomputer Interfacing,* by J.J. Carr, Reston, VA: Reston Publishing Company, 1983).

The connector used to mate with the T/S-1000 interface connector is not a standard form of connector, and is not widely available in the United States. A substitute can be made, however, if you are handy with elementary tools. At least a jeweler's file and a fine-tooth coping saw are needed. In my case, I made the connector using a Dremel moto-tool with the 1-inch

grindstone cutters. The T/S-1000 connector is a 23/46 style, meaning that it has 23 pins on each side for a total of 46 pins. You will be able to obtain from Radio Shack and other outlets 25/50 printed circuit card connectors (remember the 0.100 inch centers!), and these can be cut down. The first step will be removal of the two ends of the connector. This is necessary because the T/S-1000 interface connector is not offset from the main printed circuit board, but is flush with it. Next, remove the metal contacts for one pin on either end of the connector. Also remove the contacts for the #3 pin (which is the slot).

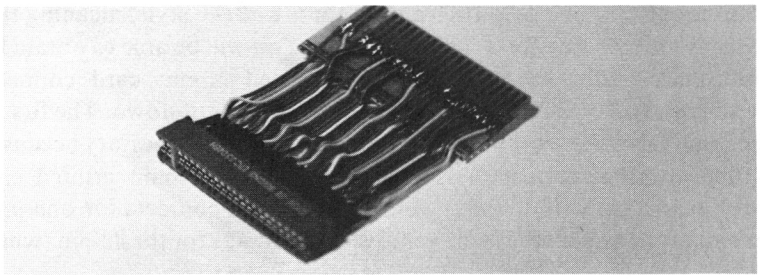
If you want to breadboard your own accessories for the T/S-1000, then it is possible to use the Vector (and other) blank or "DIP-pattern" printed cards designed for the Apple-II microcomputer. These cards are equipped with a pair of 0.100-inch centers, one of which contains the 25/50 pattern needed for 100 percent interfacing with the T/S-1000. The other connector is too short, but can be used where not all of the T/S-1000 interface connector signals are needed.

Many users of the T/S-1000 report that the computer programs "bomb out" if there is any motion between the 16K memory module, and the computer. Some users secure the 16K module to the computer with Velcro strips, while others build a little extension harness such as shown in Fig. 11-2. Although this connector is a little crude, it works.

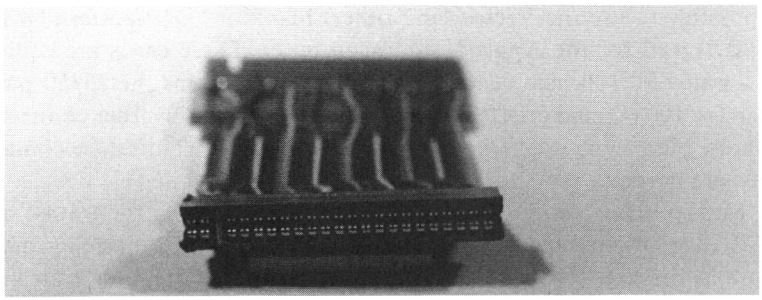
The male end of the connector is fashioned from a Radio Shack wiring board (several of these come in 0.100 inch centers, 25/50, and are cheap enough to chop up for this application without causing tears). Keep in mind that the slot must be cut into the connector in order to prevent accessories from being connected backwards (with disastrous effect). They keyway needed to insert into the female connector, the keyway which mates with the slot, can also be made from the board from which the connector is salvaged. A little judicious use of a jeweler's file will work wonders in this respect.

If you do not want to fool with the making of connectors, and cannot find commercial connectors, then you might want to buy one of the I/O port accessories available and interface everything through an I/O port—it's usually simpler!

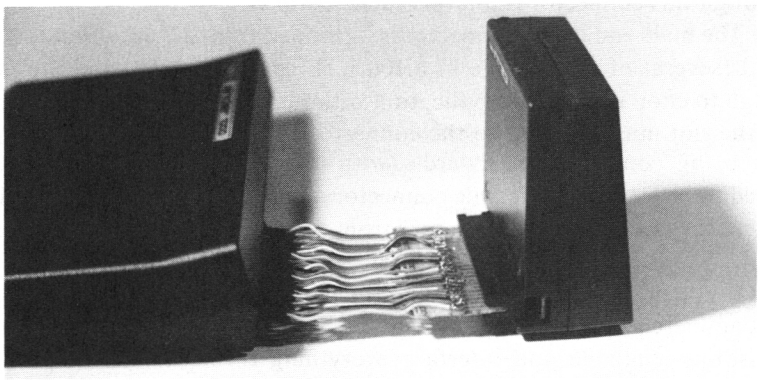
The Time 2068 is a sophisticated computer which has a powerful interface capability. There are four interface connectors on the 2068, of three different types. One of these connectors is the standard interface connector on the rear apron of the microcomputer. This connector is similar to, but larger than, the interface connector used on the 1000 and 1500 machines. The second form of connector is the DOCK connector on the front of the computer, under a lift-up cover. This connector is designed to accept the external read only memory (ROM) Command Cartridges. Finally, there are two 9-pin D-style game connectors of the sort found on *Atari* microcomputers. All of these connectors are defined on the following pages.



(a)



(b)



(c)

FIGURE 11-2 (a) Interface extension connector. (b) Homemade connector showing key-way insert. (c) Interface extension connector in place.

DOCK CONNECTOR

The dock connector is located on the front of the T/S-2068 microcomputer. It is on the righthand side, underneath a lift-up cover. This 36-pin connector uses the same 0.100-inch centers as used on the older 1000 and 1500 connectors.

Like those other connectors, there is a keyway to prevent upside-down insertion of the Command Cartridge.

The Command Cartridge contains an 8K read only memory, which can be used to hold languages other than BASIC, and applications software. There are two different forms of ROM program. The Language ROM Oriented Software (LROS) is for languages, and is mapped into locations 0000H through 5FFFH. The Applications ROM Oriented Software (AROS) locations are 8000H to FFFFH.

Timex has available a *Timex Sinclair 2000 Third Party Software Guide* which gives additional details regarding specifications needed to prepare software for the Command Cartridges. As of this writing, however, the manual is regarded as confidential by the Timex people, and can only be obtained by serious software developers who write to them requesting a copy. Hopefully, future editions of this book may contain some of that data, but for the present time the author and publishers are under a Nondisclosure Agreement.

The DOCK connector pinouts are defined below, and in Fig. 11-3. Note that the connector uses an unusual numbering scheme. The even numbered pins are on the top or "component" side of the printed wiring board, while the odd numbered pins are on the bottom or "noncomponent" side of the board.

DOCK CONNECTOR PIN-OUTS

1	A14B	Address bus bit-14 (buffered)
2	+5V	+5 Volt DC power supply connection
3	A12	Address bus bit-12
4	A13B	Address bus bit-13 (buffered)
5	D0	Data bus bit-0
6	D7	Data bus bit-7
7	D1	Data bus bit-1
8	A0	Address bus bit-0
9	D2	Data bus bit-2
10	A1	Address bus bit-1
11	D6	Data bus bit-6
12	A2	Address bus bit-2
13	D5	Data bus bit-5
14	A3	Address bus bit-3
15	D3	Data bus bit-3
16	A15	Address bus bit-15
17	D4	Data bus bit-4
18	$\overline{\text{MREQB}}$	Memory Request (buffered), active-LOW
19	$\overline{\text{IORQB}}$	Input/Output Request (buffered), active-LOW
20	A7RB	Refresh address bit-7 (buffered)
21	$\overline{\text{RDB}}$	Read (buffered), active-LOW
22	$\overline{\text{M1}}$	M1 Machine Cycle Flag — CPU (buffered), active-LOW

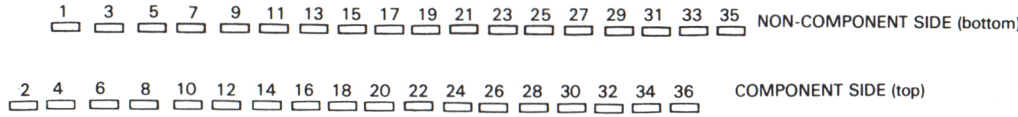


FIGURE 11-3. DOCK connector pinouts.

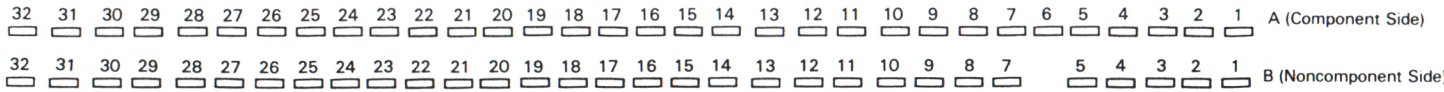


FIGURE 11-4. The interface connector pinouts.

23	$\overline{\text{WRB}}$	Write (buffered), active-LOW
24	A8	Address bus bit-8
25	A7	Address bus bit-7
26	A9	Address bus bit-9
27	A6	Address bus bit-6
28	A10	Address bus bit-10
29	A5	Address bus bit-5
30	A11	Address bus bit-11
31	A4	Address bus bit-4
32	$\overline{\text{RFSHB}}$	Refresh (buffered), active-LOW
33	$\overline{\text{BE}}$	Bank Enable, active-LOW
34	$\overline{\text{EXROM}}$	Extension Read Only Memory, active-LOW
35	$\overline{\text{ROSCS}}$	ROM Oriented Software Chip Select, active-LOW
36	GRND	GROUND (signal and power supply common)

INTERFACE CONNECTOR

The Interface Connector on the rear apron of the T/S-2068 microcomputer is similar to, but considerably larger than the interface Connector on the T/S-1000 and T/S-1500 machines. Like those other machines, the T/S-2068 connector uses 0.100-inch centers, and a keyway slot. This connector is defined below and in Fig. 11-4.

COMPONENT SIDE PINOUTS (SIDE "A"):

PIN#	SIGNAL	DESCRIPTION
1A	Tape/Spkr Out	Tape & Speaker Output
2A	DZOUT	Daisy Out
3A	A7RB	Refresh Address bit-7 (buffered)
4A	D7	Data bus bit-7
5A	(not used)	
6A	(slot)	
7A	D0	Data bus bit-0
8A	D1	Data bus bit-1
9A	D2	Data bus bit-2
10A	D6	Data bus bit-6
11A	D5	Data bus bit-5
12A	D3	Data bus bit-3
13A	D4	Data bus bit-4
14A	$\overline{\text{INT}}$	Interrupt Request, active-LOW
15A	$\overline{\text{NMI}}$	Nonmaskable Interrupt request, active-LOW

PIN#	SIGNAL	DESCRIPTION
16A	$\overline{\text{HALT}}$	CPU Halt Signal, active-LOW
17A	$\overline{\text{MREQB}}$	Memory Request (buffered), active-LOW
18A	$\overline{\text{IORQB}}$	Input/Output Request (buffered), active-LOW
19A	$\overline{\text{RDB}}$	Read (buffered), active-LOW
20A	$\overline{\text{WRB}}$	Write (buffered), active-LOW
21A	$\overline{\text{BUSAK}}$	Bus Acknowledgment, active-LOW
22A	$\overline{\text{WAIT}}$	CPU Wait, active-LOW
23A	$\overline{\text{BUSRQ}}$	Bus Request, active-LOW
24A	$\overline{\text{RESET}}$	CPU Reset, active-LOW
25A	$\overline{\text{M1}}$	M1 Machine Cycle Indicator, active-LOW
26A	$\overline{\text{RFSHB}}$	Refresh signal (buffered), active-LOW
27A	$\overline{\text{EXROM}}$	Extension ROM Enable, active-LOW
28A	$\overline{\text{ROSCS}}$	ROM Oriented Software Chip Select, active-LOW
29A	$\overline{\text{BE}}$	Bank Enable, active-LOW
30A	GRND	Ground (signal and power supply common)
31A	SOUND	Analog Sound Signal Output (0 to 5 volts)
32A	GRND	Ground (signal and power supply common)

NON-COMPONENT SIDE PINOUTS (SIDE "B"):

1B	EAR	"Ear" input
2B	DZIN	Daisy input
3B	+15V	+15 volt Power Supply
4B	+5V	+5 volt Power Supply
5B	(not used)	
6B	(slot)	
7B	GRND	Ground (signal and power supply common)
8B	GRND	Ground (signal and power supply common)
9B		CPU clock (inverted)
10B	A0	Address bus bit-0
11B	A1	Address bus bit-1
12B	A2	Address bus bit-2
13B	A3	Address bus bit-3
14B	A15B	Address bus bit-15 (buffered)
15B	A14B	Address bus bit-14 (buffered)
16B	A13B	Address bus bit-13 (buffered)
17B	A12	Address bus bit-12
18B	A11	Address bus bit-11
19B	A10	Address bus bit-10
20B	A9	Address bus bit-9

PIN#	SIGNAL	DESCRIPTION
21B	A8	Address bus bit-8
22B	A7	Address bus bit-7
23B	A6	Address bus bit-6
24B	A5	Address bus bit-5
25B	A4	Address bus bit-4
26B	(not used)	
27B	R	RED color signal
28B	G	GREEN color signal
29B	B	BLUE color signal
30B	GRND	Ground (signal and power supply ground)
31B	VIDEO	Composite video output
32B	GRND	Ground (signal and power supply ground)

JOYSTICK/GAME SIGNAL CONNECTOR

There are two game connectors on the T/S-2068 microcomputer. These connectors are standard, 9-pin "*Atari*"-style connectors in a D-type shell. These connectors are summarized below and in Fig. 11-5.

PIN #	SIGNAL	DESCRIPTION
1	$\overline{\text{DIR1}}$	Direction-1, active-LOW
2	$\overline{\text{DIR2}}$	Direction-2, active-LOW
3	$\overline{\text{DIR3}}$	Direction-3, active-LOW
4	$\overline{\text{DIR4}}$	Direction-4, active-LOW
5	(not used)	
6	$\overline{\text{BUTTON}}$	Button Input, active-LOW
7	(not used)	
8	RDSTB	Read Strobe
9	(not used)	

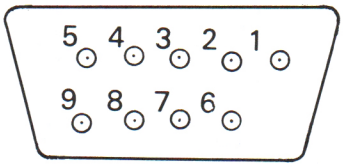


FIGURE 11-5.

OTHER T/S-2068 CONNECTORS

The T/S-2068 microcomputer uses the same connections to power and cassette players as the other Timex microcomputer: MIC, EAR, TV and Power Supply. In addition, the T/S-2068 also has a MONITOR output. These are defined as follows:

MIC	Microphone connection to the MIC jack on the cassette player.
EAR	Earphone connection to EAR jack on the cassette player.
TV	Television output. RF + composite video on regular TV channels 2 or 3 (selectable by bottom panel switch)
POWER	Power supply. *Do not* substitute other than the factory-supplied power supplies without expert knowledge!
MONITOR	This is a TV output, but not RF. This output goes to a closed-circuit TV monitor for high resolution display.

CHAPTER 12

DATA CONVERSION: A/D-D/A

WHAT ARE DATA CONVERTERS?

Analog circuits and digital instruments occupy mutually exclusive realms. In the analog world, a signal may vary between upper and low limits, and may assume any value within the range. Analog signals are continuous between limits. But the signals in digital circuits may assume only one of two discrete voltage levels, i.e., one each for the two binary digits, 0 and 1. A *data converter* is a circuit or device that examines signals from one of these realms and then converts it to a proportional signal from the other.

A *digital-to-analog converter* (DAC), for example, converts a digital (i.e., binary) "word" consisting of a certain number of *bits* into a *voltage* or *current* that represents the binary number value of the digital word. An eight-bit DAC, for example, may produce an output signal of 0 volts when the binary word applied to its digital inputs is 00000000_2, and (say), 2.56 volts when the digital inputs see a word of 11111111_2. For binary words applied to the inputs, then, a proportional output voltage is created.

In the case of an *analog-to-digital converter* (ADC, or A/D), an analog voltage or current produces a proportional binary word output. If an 8-bit ADC has a 0- to 2.56-volt input signal range, then 0 volt input could produce an output word of 00000000_2, while +2.56-volt level seen at the input would produce an output word of 11111111_2.

Data converters are used primarily to interface transducers (most of which produce analog output signals) to digital instruments or computer inputs, and to interface digital instrument outputs to analog-world devices such as meter movements, chart recorders, motors, etc.

BINARY RESISTANCE LADDER CIRCUITS

Figure 12-1 shows a *binary weighted resistance ladder* and operational amplifier used as a binary DAC. The operation of this circuit can be deduced from operational amplifier theory.

The resistors in the ladder are said to be binary weighted, because their values are related to each other by powers of two. If the lowest value resistor is given the value R, then the next in the sequence will have a value $2R$, followed by $4R$, $8R$, $16R$, all the way up to the nth resistor (last one in the chain), which has a value of $(2^{(n-1)})R$.

The switches ($B1$ through B_n) represent the input bits of the digital word. Although shown here as mechanical switches, they would be transistor switches in actual practice. The switches are used to connect the input resistors either to ground, or voltage source E, to represent binary states 0 and 1,

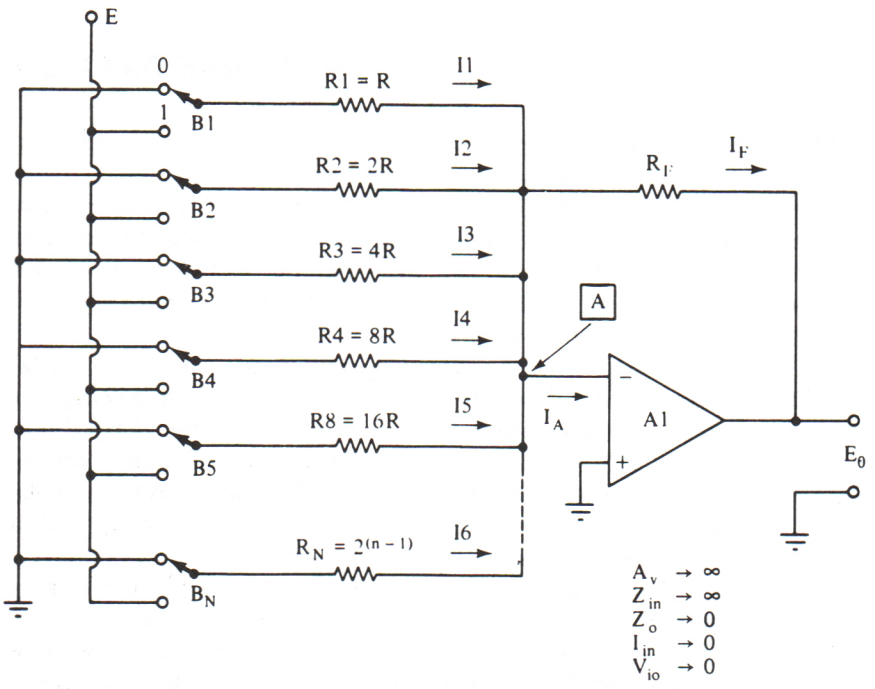


FIGURE 12-1. Binary weighted resistor ladder DAC circuit.

respectively. Switches $B1$ through B_n create currents $I1$ through I_n, respectively, when they are set to the "1" position.

We know from Ohm's law that each current $I1$ through I_n is equal to the quotient of E and the value of the associated resistor, i.e.:

$$I1 = E/R1 = E/R$$
$$I2 = E/R2 = E/2R$$
$$I3 = E/R3 = E/4R$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$I_n = E/R_n = E/(2^{(n-1)}R$$

The total current into the junction (point A in Figure 12-1) is expressed by the summation of currents $I1$ through I_n:

$$I_A = \sum_{i=1}^{n} \frac{a_i E}{2^{(i-1)}R} \tag{12-1}$$

where I_A = the current into the junction (point A) in amperes (A)

E = the reference potential in volts (V)

R = the resistance of $R1$ in ohms (Ω)

a_i = either "1" or "0," depending upon whether the input bit is "1" or "0"

n = the number of bits, i.e., the number of switches

From operational amplifier theory we know that

$$I_A = -I_f \tag{12-2}$$

and

$$E_0 = I_f R_f \tag{12-3}$$

So, by substituting Equation (12-2) into Equation (12-3), we obtain,

$$E_0 = -I_A R_f \tag{12-4}$$

and substituting Equation (12-1) into Equation (12-4) yields

$$E_0 = -R_f \sum_{i=1}^{n} \frac{a_i E}{2^{(i=1)}R} \tag{12-5}$$

Since E and R are constants, we usually write Equation (12-5) in the form

$$E_0 = \frac{-ER_f}{R} \sum \frac{a_i}{2^{(i-1)}R} \tag{12-6}$$

Example 12-1

A four-bit (i.e., $n = 4$) DAC using a binary weighted resistor ladder has a reference source of 10 volts DC, and $R_f = R$. Find the output voltage E_0 for the input word 1011_2. (*Hint:* for input 1011, $a_1 = 1$, $a_2 = 0$, $a_3 = 1$, and $a_4 = 1$.)

Solution

$$E_0 = \frac{-ER_f}{R} \sum \frac{a_i}{2^{(i-1)}R} \tag{12-6}$$

$$E_0 = \left[\frac{(-10\text{ V})(R)}{(R)}\right]\left[\frac{1}{2^{(1-1)}} + \frac{0}{2^{(2-1)}} + \frac{1}{2^{(3-1)}} + \frac{1}{2^{(4-1)}}\right]$$

$$E_0 = (-10\text{ V})\left[\frac{1}{2^0} + \frac{1}{2^2} + \frac{1}{2^3}\right]$$

$$E_0 = (-10\text{ V})\left[\frac{1}{1} + \frac{1}{4} + \frac{1}{8}\right]$$

$$E_0 = (-10\text{ V})\left[\frac{1}{1.375}\right] = -\mathbf{7.27\text{ volts}}$$

R-2R RESISTANCE LADDER CIRCUITS

Although not revealed by the idealized equations, the binary weighted resistance ladder suffers from a serious drawback in actual practice. The values of the input resistors tend to become very large and very small at the ends of the range as the bit length of the input word becomes longer. If R is set to 10 kΩ (a popular value), then $R8$ will be 1.28 megohms. If we assume a reference potential E of 10.00 volts DC, then $I8$ will be only 7.8 *microamperes.* Most common nonpremium-grade opertional amplifiers will not be able to resolve signals that low from the inherent noise. As a result, the bit length of the binary weighted ladder is severely limited. Few of these types of converters are found with more than 6- or 8-bit word lengths.

In commercial DACs, all of the resistors have a value of either R or $2R$ (Fig. 12-2). The gain of the amplifier is unity, so E_0 can be expressed as

$$E_0 = (E)\sum_{i=1}^{n} \frac{a_i}{2_i} \tag{12-7}$$

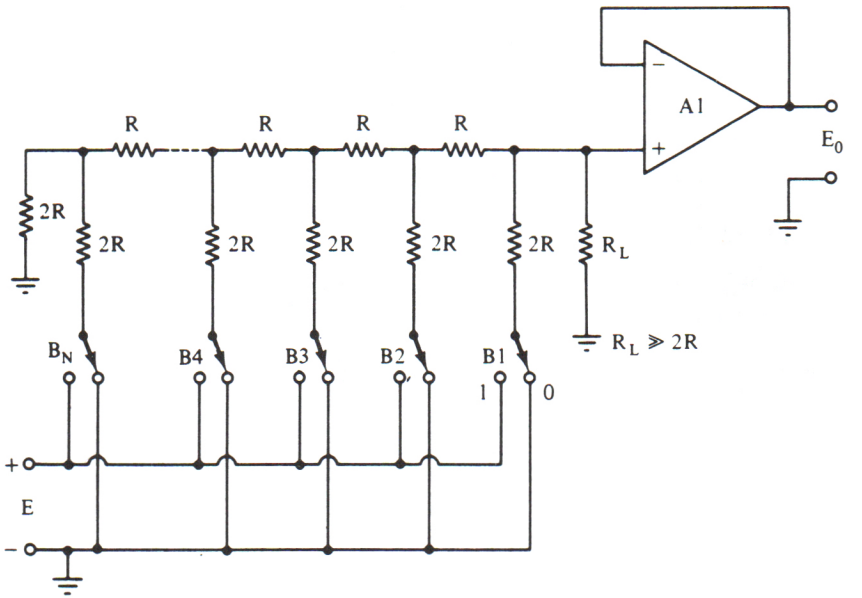


FIGURE 12-2. R-2R resistor ladder DAC circuit.

(provided that $R_L >> R$, so that the voltage divider effect between the ladder and R_L can safely be neglected).

Example 12-2

A four-bit DAC using the R-$2R$ technique has a 5.00-volt DC reference potential. Calculate E_0 for the input word 1011_2.

Solution

$$E_0 = E \sum_{i=1}^{n} \frac{a_i}{2^i} \tag{12-7}$$

$$E_0 = (5\ \text{V}) \left[\frac{1}{2^1} + \frac{0}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} \right]$$

$$E_0 = (5\ \text{V}) \left[\frac{1}{2} + 0 + \frac{1}{8} + \frac{1}{16} \right]$$

$$E_0 = (5\ \text{V})(0.688) = \mathbf{3.44\ volts}$$

The full-scale output voltage for any DAC using the R-$2R$ resistor ladder is given by

$$E_{fs} = \frac{E(2^n - 1)}{2^n} \tag{12-8}$$

where E_{fs} = the full-scale output potential in volts (V)
E = the reference potential in volts (V)
n = the bit length of the digital input word

Example 12-3

Find the full-scale potential for an eight-bit DAC with a reference potential of 10.00 volts DC.

Solution

$$E_{fs} = \frac{E(2^n - 1)}{2^n} \tag{12-8}$$

$$E_{fs} = \frac{(10\text{ V})(2^8 - 1)}{2^8}$$

$$E_{fs} = \frac{(10\text{ V})(2^7)}{(2^8)}$$

$$E_{fs} = \frac{(10\text{ V})(255)}{(256)} = \mathbf{9.96\ volts}$$

The output of a DAC cannot change in a continuous manner, because the input is a digital word; i.e., it can exist only in certain discrete states. Each successive binary number changes the output an amount equal to the change created by the least significant bit (LSB), which is expressed by

$$\Delta E_0 = \frac{E}{2^n} \tag{12-8}$$

So, for the DAC in Example 12-3, E_0 would be

$$\Delta E_0 = \frac{(10\text{ V})}{2^8}$$

$$\Delta E_0 = \frac{(10\text{ V})}{(256)} = \mathbf{40\ mV}$$

ΔE_0 is often called the 1 LSB value of E_0, and is the smallest change in output voltage that can occur. It is interesting that, if we let 0 volts represent 00000000_2 in our eight-bit system, then the maximum value of E_0 at 11111111_2 will be 1 LSB less than E (confirmed by the result of Example 12-3.)

There are numerous commercial DACs on the market in IC, function module block, and equipment form. The reader should consult manufacturer's catalogues for appropriate types in any given application.

BASICS OF A/D CONVERSION

The purpose of the analog-to-digital converter is to produce an output binary word that is proportional to an analog input signal. The output of the A/D converter is input to a computer, where it is used in the various processes and calculations made by the computer. The type of A/D converter selected often depends upon the job that must be done, so sufficient concern must be shown for the specifications.

TYPES OF A/D CONVERTERS

There are several different *basic* analog-to-digital converter circuits, and they all have their own typical applications. Although the circuitry varies from one model to another, it is often the *speed* that is the most important specification in some applications, cost being second in importance. The *dual-slope integrator* is often very low in cost but is very slow. Similarly, the *flash* converter is very rapid (video rates), but it is difficult to decode into a format that the computer likes, and must often be used with RF-like connecting circuitry. The *successive approximation* converter usually acquits itself nicely, except that it can be too complex and costly. It will, however, easily outperform most versions of the other types of converter.

In this book, we will consider only these basic converter types: *servo* (also called *binary ramp*), *successive approximation*, *flash* (or *parallel*), *dual-slope integration*, and *voltage-to-frequency converters.*

SERVO ADC CIRCUITS

The *servo* ADC circuit (also called *binary counter* or *ramp* ADC circuit) uses a binary counter to drive the digital inputs of a DAC. A voltage comparator keeps the clock gate to the counter open as long as $E_0 \neq E_{in}$.

An example of such a circuit is the eight-bit ADC in Figure 12-3(a), while the relationship of E_0 and E_{in} relative to time is shown in Figure 12-3(b).

Two things happen when a *start* pulse is received by the control logic circuits: the binary counter is reset to 00000000_2, and the gate is opened to allow clock pulses into the counter. This will permit the counter to begin incrementing, thereby causing the DAC output voltage E_0 to begin rising,

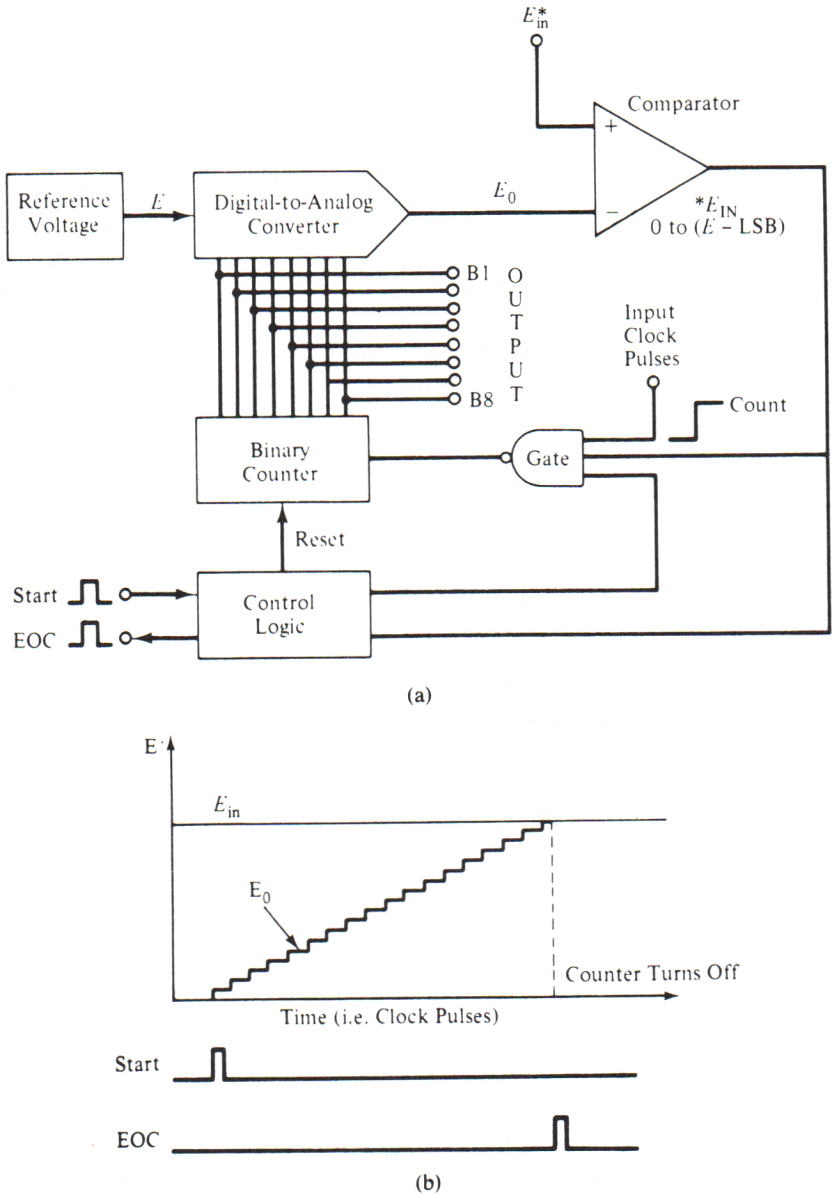


FIGURE 12-3. (a) Servo ADC circuit. (b) Operation of the servo-type ADC circuit.

Figure 12-3(b). E_0 will continue to rise until $E_0 = E_{in}$. When this condition is met, the output of the comparator drops *low*, turning off the gate. The binary number appearing on the counter output at this time is proportional to E_{in}.

The control logic section senses the change in comparator output level, and uses it to issue an *end-of-conversion* (EOC) pulse. This EOC pulse is used

by instruments or circuitry connected to the ADC to verify that the output data are valid.

The *conversion time* T_c of an ADC such as this depends upon the value of E_{in}, so when E_{in} is maximum (i.e., full scale), so is T_c. Conversion time for this type of ADC is on the order of 2^n clock pulses for a full-scale conversion.

SUCCESSIVE APPROXIMATION ADC CIRCUITS

The conversion time of the servo ADC is too long for some applications. The successive approximation (SA) ADC is much faster for the same clock speed; i.e., it takes $(n + 1)$ clock pulses instead of 2^n. For the eight-bit ADC that has been our example, the SA type of ADC is 28 times *faster* than the *servo* ADC.

The basic concept of the SA ADC circuit can be represented by a platform balance, such as Figure 12-4, in which a full-scale weight W will deflect the pointer all the way to the left when pan 2 is empty.

Our calibrated weight set consists of many separate pieces, which weigh $W/2$, $W/4$, $W/8$, $W/16$, etc. When an unknown weight W_x is placed on pan 2, the scale will deflect to the right. To make our measurement, we start with $W/2$, and place it on pan 1. Three conditions are now possible:

$W/2 = W_x$ (scale is at zero)

$W/2 > W_x$ (scale is to the left of zero)

$W/2 < W_x$ (scale is to the right of zero)

If $W/2 = W_x$, then the measurement is finished, and no additional trials are necessary. But if $W/2$ is less than W_x, then we must add more weights in succession ($W/4$, then $W/8$, etc.) until we find a combination equal to W_x.

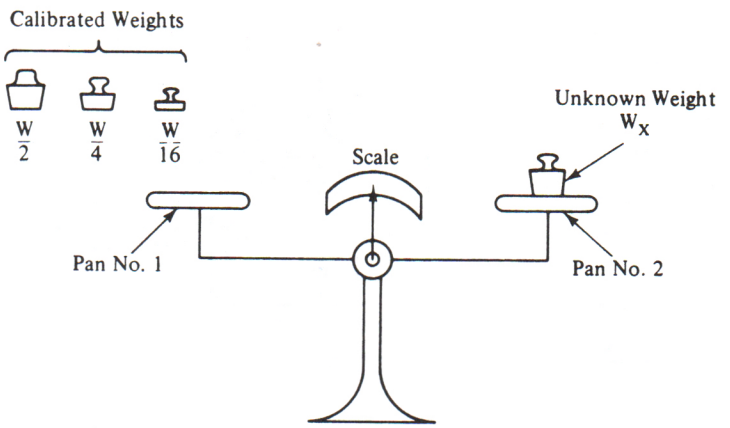


FIGURE 12-4. Successive approximation ADC circuits are like the platform balance.

If, on the other hand, $W/2$ is greater than W_x, then we must *remove* the $W/2$ weight, and in the second trial start again with $W/4$. This procedure will continue until a combination equal to W_x is found.

In the SA ADC circuit we do not use a scale, but a shift register, as in Figure 12-5. A successive approximation register (SAR) contains the control logic, a shift register, and a set of output latches, one for each register section. The outputs of the latches drive a DAC.

A *start* pulse sets the first bit of the shift register *high*, so the DAC will see the word 10000000_2, and therefore produces an output voltage equal to one-half of the full-scale output voltage. If the input voltage is greater than $\frac{1}{2}E_{fs}$, then the $B1$ latch is set *high*. On the next clock pulse, register $B2$ is set high for trial 2. The output of the DAC is now $\frac{3}{4}$-scale. If, on any trial, it is found that $E_{in} < E_0$, then that bit is reset *low*.

Let us follow a three-bit SAR through a sample conversion. In our example, let us say that the full-scale potential is 1 volt, and E_{in} is 0.625 volt. Consider Figure 12-6.

Time t_1: The *start* is received, so register $B1$ goes *high*. Output word is now 100_2, so $E_0 = 0.5$ V. Since E_0 is less than E_{in}, latch $B1$ is set to "1," so at the end of the trial, the output word remains 100_2,

Time t_2: On this trial (which starts upon receiving the next clock pulse), register $B2$ is set *high*, so the output word is 110_2.

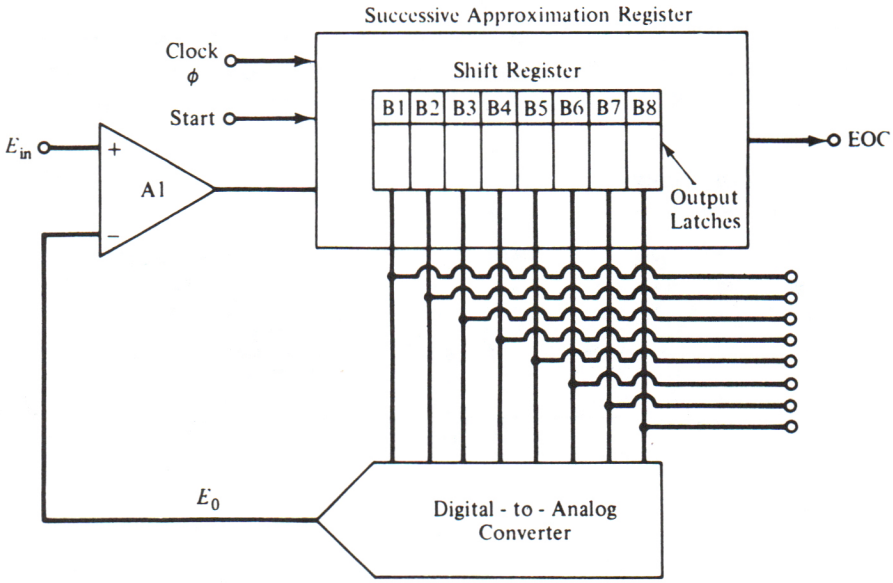


FIGURE 12-5. Successive approximation ADC circuit.

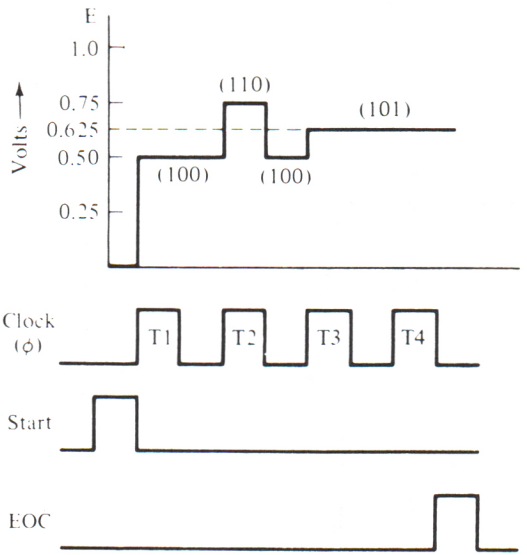


FIGURE 12-6. Timing diagram for Figure 12-5.

Voltage E_0 is now 0.75 volt. Since E_{in} is less than E_0, the $B2$ latch is set to "0," and the output word reverts to 100_2.

Time t_3: Register $B3$ is set *high*, making the output word 101_2. The value of E_0 is now 0.625 volt, so $E_{in} = E_0$. The $B3$ register is latched to 1, and the output word remains 101_2.

Time t_4: Overflow occurs, telling the control logic to issue an EOC pulse. In some cases the overflow pulse *is* the EOC pulse.

Note that in the example, we had a three-bit SAR, so by our $(n + 1)$ rule, required four clock pulses to complete the conversion. The SA type of ADC was once regarded as difficult to design because of the logic required. But today IC and function blocks are available that use this technique, so the design job is reduced considerably. The SA technique can be implemented in software under computer control using only an external DAC and comparator. All register functions are handled in the software (program).

PARALLEL CONVERTERS

The parallel ADC circuit (Figure 12-7) is probably the fastest type of ADC known. In fact, some texts call it the "flash" converter in testimony to its speed. It consists of a bank of $(2^n - 1)$ voltage comparators biased by reference potential E through a resistor network that keeps the individual comparators 1 LSB apart. Since the input voltage is applied to all of the

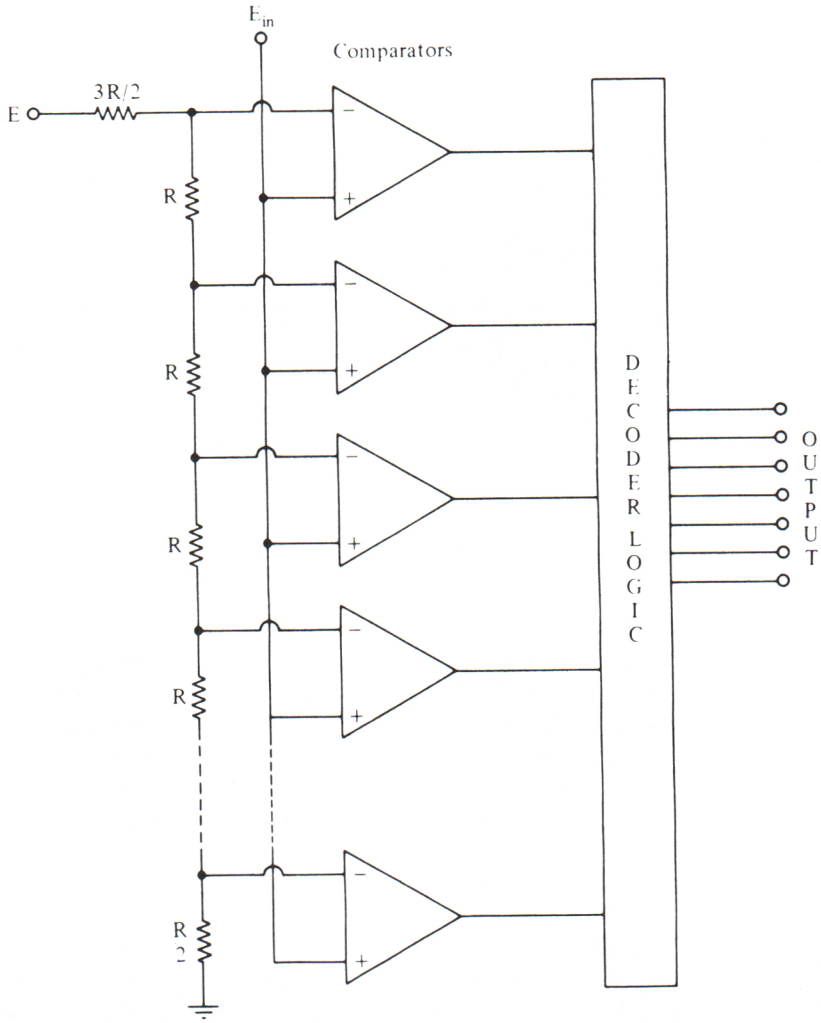


FIGURE 12-7. Parallel or "flash" ADC circuit.

comparators simultaneously, the speed of conversion is essentially the slewing speed of the *slowest* comparator in the bank, and the decoder propagation time (if logic is used). The decoder converts the output code to binary code, or possibly BCD in some cases.

VOLTAGE-TO-FREQUENCY CONVERTERS

A voltage-to-frequency (V/F) converter is a voltage-controlled oscillator (VCO) in which an input voltage E_{in} is represented by an output frequency F. An ADC using the V/F converter is shown in Figure 12-8. It consists of no

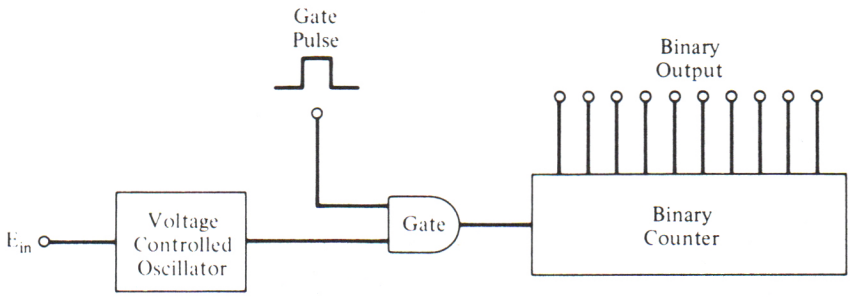


FIGURE 12-8. Voltage-to-frequency converter ADC.

more than the VCO and a frequency counter. The display, or output states, on the counter gives us the value of E_{in}.

Voltage-to-frequency converters are used mainly where economics dictate serial transmission of data from a remote collection point to the instrument. Such data can be transmitted by wire or radio communications channels. Another application is the tape recording of analog data that is, in itself, too low in frequency to be recorded.

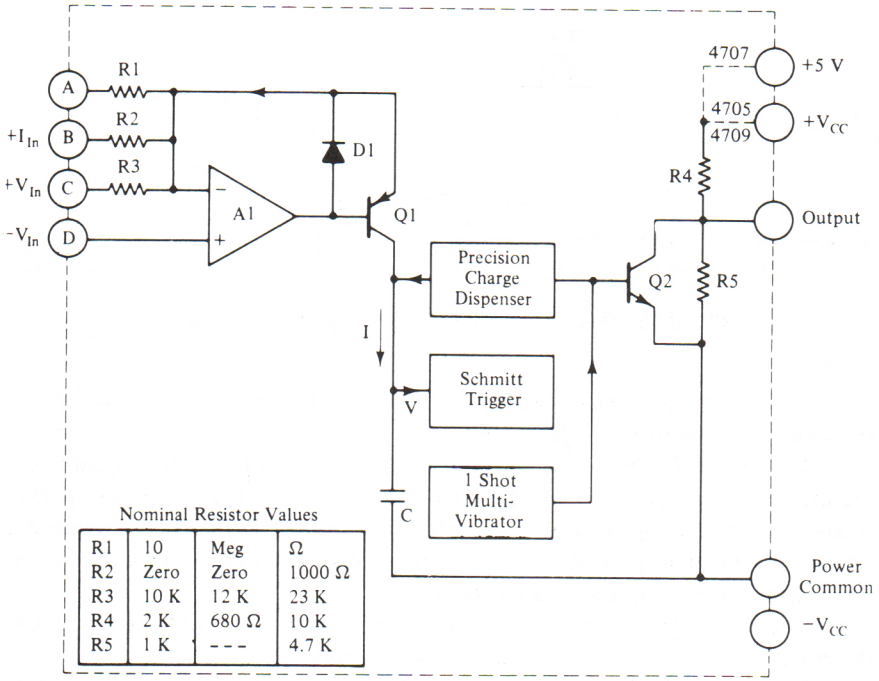
The inverse procedure, F/V conversion, is a form of DAC, in which an input *frequency* is converted to an output *voltage*.

The type of V/F converter shown in Figure 12-8 is a simplistic circuit and generally too crude for practical application. Most commercial V/F converters are variations on the circuit shown in Figure 12-9, which is a version of a product made by Teledyne/Philbrick.

The V/F converter is basically a freerunning astable multivibrator in which the operating frequency is controlled by the input voltage. This circuit may also be called a *voltage controlled oscillator* (VCO), although in common application the VCO has too limited a range for serious A/D work.

The input operational amplifier (A1) is connected as a driver for the precision current source, transistor Q1. The collector current of this NPN transistor is a linear function of the input currents applied to amplifier A1. The collector current (I) is used to charge timing capacitor C. We may conclude, therefore, that the charging rate of capacitor C is also a linear function of the input currents applied to $A1$. Note that a capacitor that is charged from a *constant current source* (e.g., Q1) will have a linear ramp for the charging voltage, and *not* the exponential ramp expected for *constant voltage source* circuits that are normally considered in elementary texts.

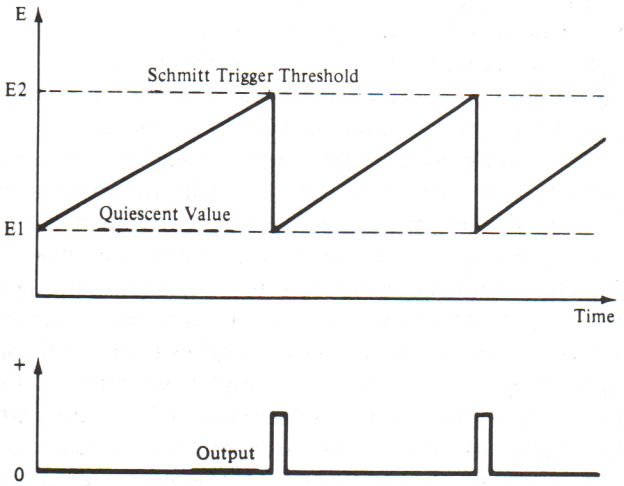
Schmitt triggers are special cases of comparator circuits in which the output snaps HIGH when the analog input voltage crosses a certain threshold in the positive-going direction, and then drops LOW again when the input voltage falls below another threshold in the negative-going direction. The capacitor voltage, E_c, is used as the analog input to the Schmitt trigger circuit in Figure 12-9(a). When the capacitor voltage reaches the trip threshold [$E2$ in Figure 12-9(b)], the Schmitt trigger will go HIGH, and (a) trigger the



Nominal Resistor Values

R1	10	Meg	Ω
R2	Zero	Zero	1000 Ω
R3	10 K	12 K	23 K
R4	2 K	680 Ω	10 K
R5	1 K	---	4.7 K

(a)



(b)

FIGURE 12-9. Voltage-to-frequency converter. (a) Circuit. (b) Waveforms.

monostable multivibrator (one-shot) that produces the V/F converter output pulse, and (b) activate the *precision charge dispenser* (PCD) that will discharge capacitor C to the same level (non-zero) each time it is turned on.

The timing waveforms for the V/F converter are shown in Figure 12-9(b). The voltage across the capacitor will rise from its quiescent, non-zero, value until it reaches $E2$. This voltage is the Schmitt trigger threshold, so it will trigger the circuit monostable multivibrator into producing an output pulse. At this same time, the PCD is also activated, causing the capacitor to discharge to the quiescent value, $E1$.

The slope of the capacitor charging waveform, hence the frequency of the monostable multivibrator, is a function of the collector current of transistor Q1, current I. This current is, in turn, a function of the analog input voltages (or currents, depending upon which input terminal is used) to amplifier A1, we can see that the output frequency is proportional to the input voltage. Typical V/F converters are commonly available with frequencies up to 10 kHz, with premium models or grades with 100 kHz frequencies. Some special units are known that will output frequencies up to 100 kHz.

DUAL-SLOPE INTEGRATION

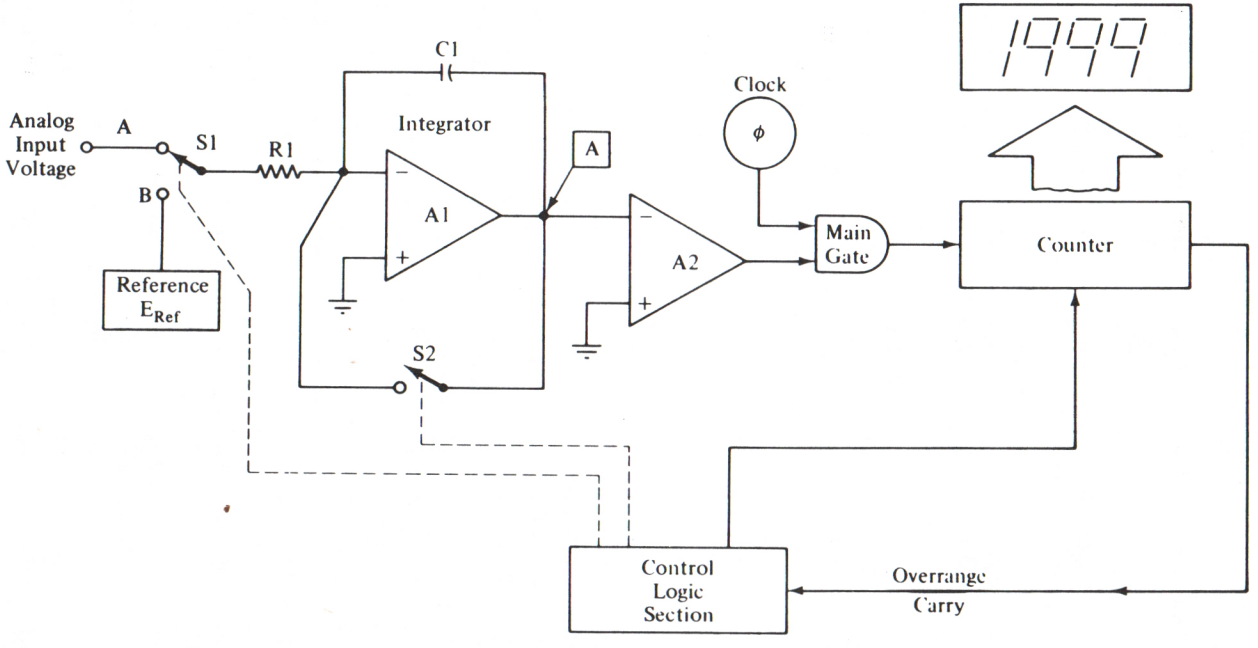
The block diagram of a dual-slop integrator is shown in Figure 12-10(a), while associated waveforms are shown in Figure 12-10(b).

The heart of the circuit is the operational amplifier *integrator* consisting of operational amplifier $A1$, plus $R1$ and $C1$, plus a *voltage comparator* ($A2$). The output of the comparator will remain LOW if the integrator output is zero, and HIGH if the integrator output is more than a few millivolts above ground potential.

At the beginning of the conversion cycle, the control logic section momentarily closes electronic switch $S2$ so that the charge on capacitor $C1$ goes to zero, and also ensures that $S1$ is set to position "A."

If $S1$ is in position "A," the integrator input is connected to the input voltage source, causing the voltage at the integrator output to begin rising [see time t_0 in Figure 12-10(b)]. As soon as E_A rises a few millivolts the comparator output snaps HIGH, enabling the gate to pass clock pulses to the digital counter section. The counter is allowed to overflow [$t1$ in Figure 12-10(b)], and the output *carry* pulse from the counter is used to tell the control logic section to switch $S1$ to position "B." This action connects the integrator input to a *precision reference* voltage source. The polarity of the input current created by the reference is such that it begins to *discharge* the integrator capacitor at a *constant* rate. The counter, meanwhile, has continued to increment, passing through 0000 at time $t1$, and continuing to accumulate clock pulses until E_A is back down to zero.

The value of E_A at time t_4 was proportional to the value of E_{in}. At the same instant the count was 0000. Since the counter continues to increment as



(a)

FIGURE 12-10. (a) Dual-slope integrator for electronic voltmeters.

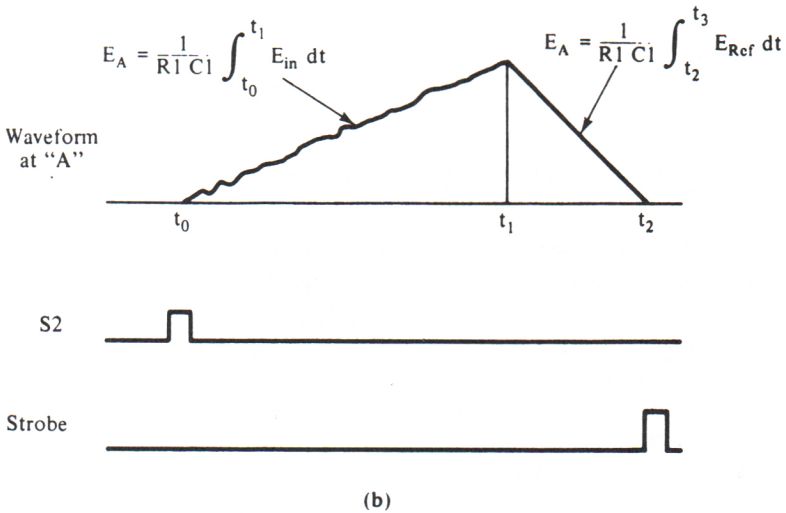


FIGURE 12-10. (b) Timing of the dual-slope integrator.

the integrator discharges (t_1 to t_2), the *count* at the instant the gate is closed (when $E_A = 0$) is also proportional to E_{in}. By correct scaling the count will be numerically the same as the potential applied to the input. To recapitulate:

1. At time t_0 switch $S2$ is closed briefly to dump any residual charge in $C1$, and $S1$ is set to position "A."
2. The integrator begins to charge due to current $E_{in}/R1$, so E_A begins rising from zero.
3. As soon as E_A is greater than zero, the output of the comparator goes HIGH, enabling the main gate to pass clock pulses into the counter.
4. The counter increments until it overflows at time t_1, and the overflow pulse causes $S1$ to switch to position "B," applying the reference voltage to the input of the integrator. At this instant the count is 0000.
5. From times t_1 to t_2 the integrator *discharges* under the influence of current $-E_{ref}/R1$; meanwhile, the counter continues to increment.
6. At time t_2 the comparator shuts off the flow of clock pulses through the gate. The count accumulated between t_1 and t_2 represents the input voltage.
7. The control logic section issues a *strobe* pulse at time t_2 to update the counter's digital display.

The dual-slope integrator is one of the slower types of ADC circuit, but offers several advantages when used in DVM circuit including relative

immunity to noise riding on the input voltage and relative immunity to error due to inaccuracy or long term drift in the clock frequency. The 10 to 40 ms conversion cycle time poses no disadvantage in most DVM applications.

WHAT DOES THE DUAL-SLOPE DVM MEASURE?

A DC digital voltmeter tends to read the *average* input voltage over the period of integration. This feature makes the DVM particularly useful for measuring noisy signals. It also tends to average the pulsations from a rectifier-type AC-to-DC converter, provided that t_2-t_0 is greater than about 18 ms.

The accuracy of a digital voltmeter, or DMM based on a DVM, can be quite good. In fact, it can exceed the accuracy of analog multimeters. You will see terms such as "$2\frac{1}{2}$-digit," "$3\frac{1}{2}$-digit," and "$4\frac{1}{2}$-digit" in the advertisements for DVM/DMM products, and will wonder what is meant by a "$\frac{1}{2}$" digit. This term refers to the fact that the *most significant* digit can only be a "0" or a "1," while all other digits can be anything between "0" and "9." Such terminology indicates that the meter can read 100 percent overrange from its basic range. For example, a $3\frac{1}{2}$-digit DVM will read 0–1999 mV, while its basic range is only 0–999 mV. If this range is exceeded, then the "1" lights up; otherwise it remains darkened.

DATA CONVERTERS AND DATA CONVERSION

Computer enthusiasts tend to believe that all jobs are best done on digital instruments . . . and to some extent this may be true. But it is a fact of life that most of nature is *analog*. The vast majority of transducers, as well as naturally "transducable" events, produce analog voltage or current signals as outputs. When, for example, we want to measure a fluid pressure (medical instruments, factory process instruments, etc.), we will find a number of different appropriate transducers that have one thing in common: the output is a DC voltage that is proportional to the applied pressure. Very few digital transducers exist. In fact, one that purportedly was "digital" was actually an analog transducer with a built-in analog-to-digital converter.

So we find that most of the instrumentation input data in analog in nature. What about output data displays? In many cases, perhaps most, we can use digital display devices for the output data. But sometimes we may require either an oscilloscope or strip-chart recorder for the display, and both of those are analog displays. In still other cases, the output data produced by the computer will be in the form of an analog voltage or current that is used to drive some prime mover, such as a motor.

We have, therefore, established clearly a need for making both analog-to-digital (A/D) conversions and digital-to-analog conversions (DAC).

Although the circuitry needed to accomplish these chores is beyond the scope of this chapter, we will discuss how these devices are interfaced with microcomputer and microprocessor instruments. Before using either A/D or DAC circuits, however, I recommend that you read a book on the subject; there are pitfalls to their use that are not immediately apparent.

Definitions

A *digital-to-analog converter* is a circuit or device that will produce either an output voltage or an output current that is proportional to the product of some reference voltage (or current) and a binary word applied to its inputs.

An *analog-to-digital converter* (A/D or ADC) is a circuit or device that will produce a binary digital word that is proportional to a reference voltage (or current) and an applied (unknown) voltage or current.

In addition to the definition for the DAC above, you will also see the term "multiplying DAC." This merely indicates that the DAC requires an external reference voltage (or current) source. The output, recall, is proportional to the *product* of the reference and the applied binary word. A nonmultiplying DAC will have the reference source built in.

DAC INTERFACING

Perhaps the easiest DAC interfacing chore is the connection of a DAC to a microcomputer output port of exactly the correct number of bits or less (there are problems when the DAC word length exceeds the microcomputer output port word length). This connection is shown in Figure 12-11. Many microcomputers have output ports in excess of their requirements, and it is often appropriate to use one of the "extras" for data conversion. In this case, it is merely necessary to connect the lines of the output port to their corresponding lines on the DAC. When a binary word is written to the output port, the word will be automatically connected to the input of the DAC, hence the DAC will produce an output voltage/current that is proportional to the binary word. This method, one of two basic methods, is called *I/O-based interfacing* of the DAC.

The method of Figure 12-11 assumes that there is an output port on the computer, and, further, that the output lines are *latched*. This is the situation that exists on most microcomputers. But oddly enough there are a few models on the market that use non-latched output ports, and these must somehow be accommodated by the instrumentation designer who (for some reason or another) must use them. Figure 12-12 shows a method for accommodating the non-latched microcomputer output port.

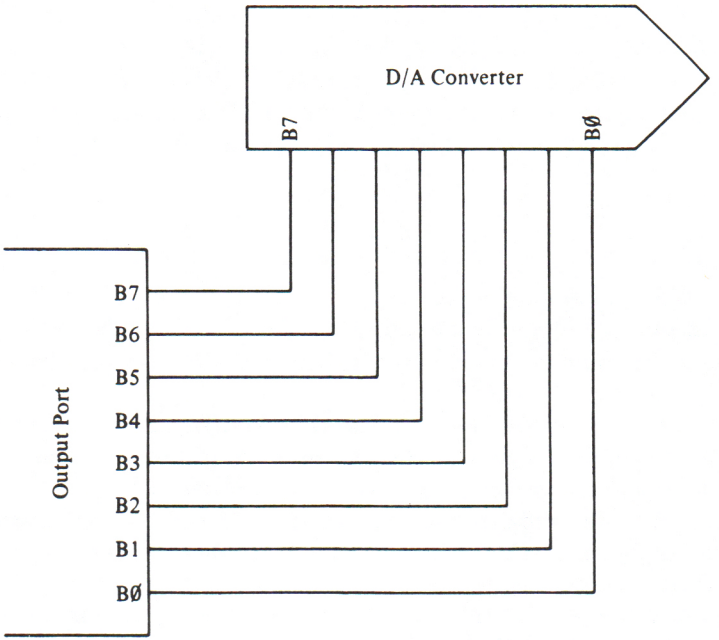


FIGURE 12-11. Interfacing DAC to output port.

In this application we are limited to a single eight-bit output port, yet certain "housekeeping" chores must be performed, i.e., data strobes and data transfer. The circuit in Figure 12-12 uses a pair of 74100 dual quad latches. These ICs contain two independent, four-bit data latches. These will transfer four-bit words on the input lines to the output lines whenever the strobe input is brought HIGH. By connecting the two strobe lines on a 74100 to a common line, we will make a single eight-bit data latch.

IC1 in Figure 12-12 is used with the two four-bit data latches independent. Notice that we connect both halves of IC1 to the same output port lines, i.e., BØ and B4 of the 74100 are connected to BØ of the output port, B1/B5 to bit B1 of the port, etc. We will need to write a program that will output half of the data word (i.e., four bits) at a time. For example, we could write the low order half byte to BØ–B3 of the output port, and, in addition, make B5 HIGH. With B5 HIGH, the data on BØ–B3 is input to BØ–B3 of the 74100. Following that operation, B5 is made LOW again, and then the high order half byte of the data word is output to BØ–B3 of the output port, and B4 of the output port is made HIGH. This condition turns on the high order strobe, so it will allow bits BØ–B3 of the output port to be input to bits B4–B7 of the 74100. All through this sequence of operations, the DAC has been responding to the data on the outputs of the other 74100 device (IC2), so it will not undergo any changes. However, once we have loaded IC1 with the entire

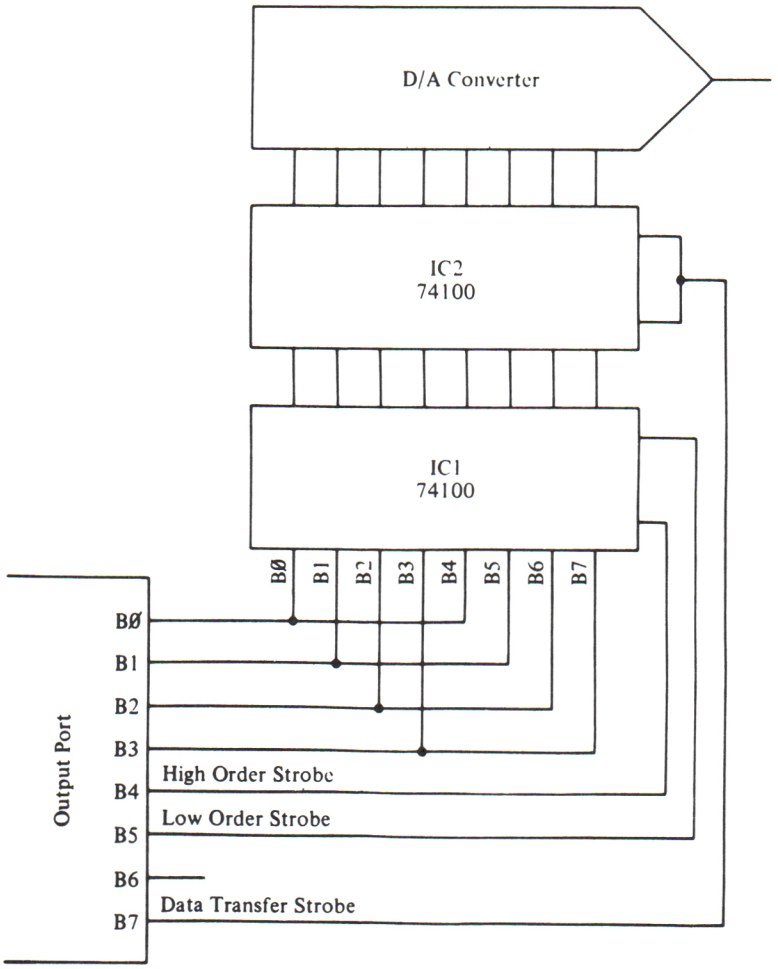


FIGURE 12-12. Double buffered system.

data word to be applied to the second 74100, then we can set B7 of the output port HIGH, causing the data that is latched onto the outputs of IC1 to be transferred to the outputs of IC2, hence to the DAC inputs. This method is called a *double-buffered latch.*

The interfacing methods thus far discussed in this chapter have been examples of I/O-based interfacing. In Figure 12-13 we see a method that could be either I/O-based, or, *memory-mapped* (also sometimes called *memory mapped-I/O*). Which it represents depends upon whether the OUT signal that controls the operation of the circuit is generated by an output command, or, a memory write command. The output device is the same 74100 that was used in previous examples. In this case, however, the input lines of the

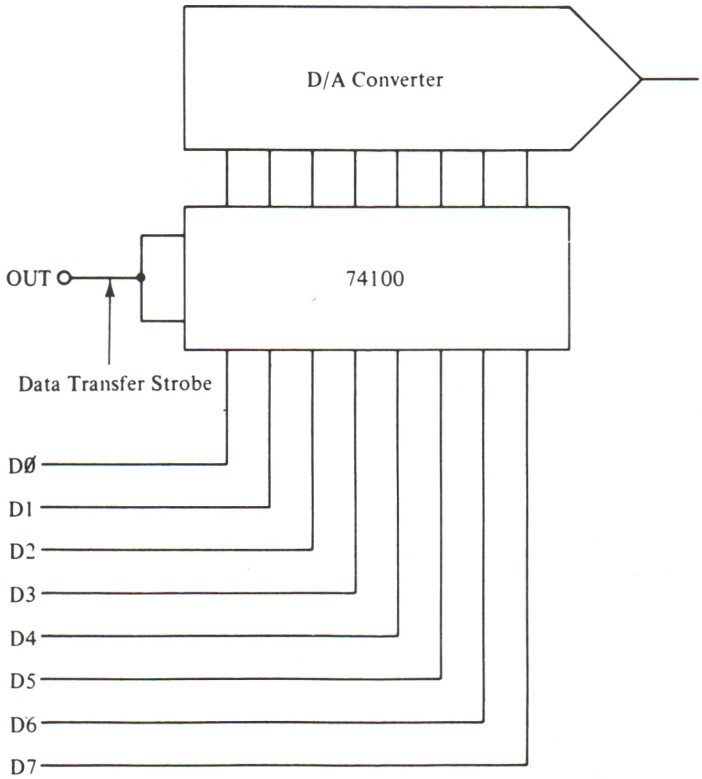


FIGURE 12-13. Interfacing DAC to data bus.

74100 are connected directly to the data bus, which is eight bits long in most of the popular microcomputers. The output lines of the 74100 data latch are connected directly to the digital input lines of the digital-to-analog converter.

If this circuit is connected as an output port, then the OUT signal will be generated in response to an output command. Some microprocessors, such as the Z80 device, have separate I/O and memory write instructions. Others, such as the 68xx and 65xx (used in the Apple-II) series, use memory locations for the output port functions; these are called memory-mapped machines. In either case, the instruction being executed will place the output data on the data bus (DØ–D7) and at the same time generate an appropriate device select (OUT) signal. When the OUT signal goes HIGH, then the 74100 will transfer data from the data bus to the DAC.

Not all DACs will require an external data latch to make these circuits work. An increasing number of devices are on the market that contain an internal data latch. One device, made by Ferranti Electric of England, has an internal latch that will also function as an internal eight-bit binary counter, depending upon the logic level applied to a control pin on the IC case.

Interfacing 10-, 12-, and 16-Bit DACs

Most microcomputers, and the microprocessor chips that form the computer, are eight-bit devices. An eight-bit data word will allow us to generate up to 256 discrete steps at the output of the DAC. In some cases, however, this does not result in sufficient resolution, so we must go to a DAC with a longer word length. For the sake of example, let us see exactly what the effect on resultion is when the data word length is manipulated. In the case of a system with a 10.00 volt full-scale output potential, the resolution will be 10/256 or about 40 mV per LSB change in the data word. In the case of a 10-bit DAC, however, the resolution is 10.00 V/(2^{10}), which is 10.00/1024, or about 9.8 mV. Similarly, 12-bit DACs produce a resolution of 2.4 mV and 16-bit DACs will resolve to 0.15 mV.

Obviously, extra bits on the DAC will result in more resolution (provided that the rest of the circuit is also upgraded—a 10-bit DAC connected to a reference source capable of supporting only an 8-bit DAC is still only an 8-bit DAC despite all the bits). But how do we make use of that extra precision in an eight-bit microcomputer. From a software point of view, we can use the double precision methods. Many of the popular microprocessor chip sets include double precision instructions. The Z80 device, for example, will allow certain instructions that make use of paired eight-bit registers to simulate the action of a 16-bit machine. In other microprocessor systems, we can write the software as double precision, treating successive eight-bit locations as if they were a single 16-bit location. Of course, this method will double our time requirements, but that is often a reasonable trade-off in instrument design.

The only question that remains, then, is how do we physically connect the 10- or more bit DAC to an eight-bit microcomputer. Figure 12-14 shows a general method that is applicable to 10- and 12-bit DACs, and by extension, to 16-bit DACs. In the 16-bit case, replace the four-bit registers (IC1 and IC3) with 74100 devices (making IC1 through IC4 all 74100s). In this discussion we are going to use the Z80 terminology because it is among the easiest microprocessor chips to program for double precision operations. Keep in mind that the eight-bit microprocessor can only accommodate eight bits of data at a time because the data bus in only 8 bits in length. In a manner similar to Figure 12-12 we are going to *multiplex* the data (10-, 12- or 16-bits) onto the data bus in successive operations. To illustrate this process, let's assign three output port locations as follows:

175_{10}	(AF_{16})	Lower 8 bits of data to DAC
176_{10}	($BØ_{16}$)	Higher (2, 4 or 8 bits) of data to the DAC
177_{10}	($B1_{16}$)	Control of output latch (IC3 and IC4)

This circuit (Figure 12-14) will first load the data into IC2, then load the remaining portion of the data into IC1, and then, simultaneously, transfer the

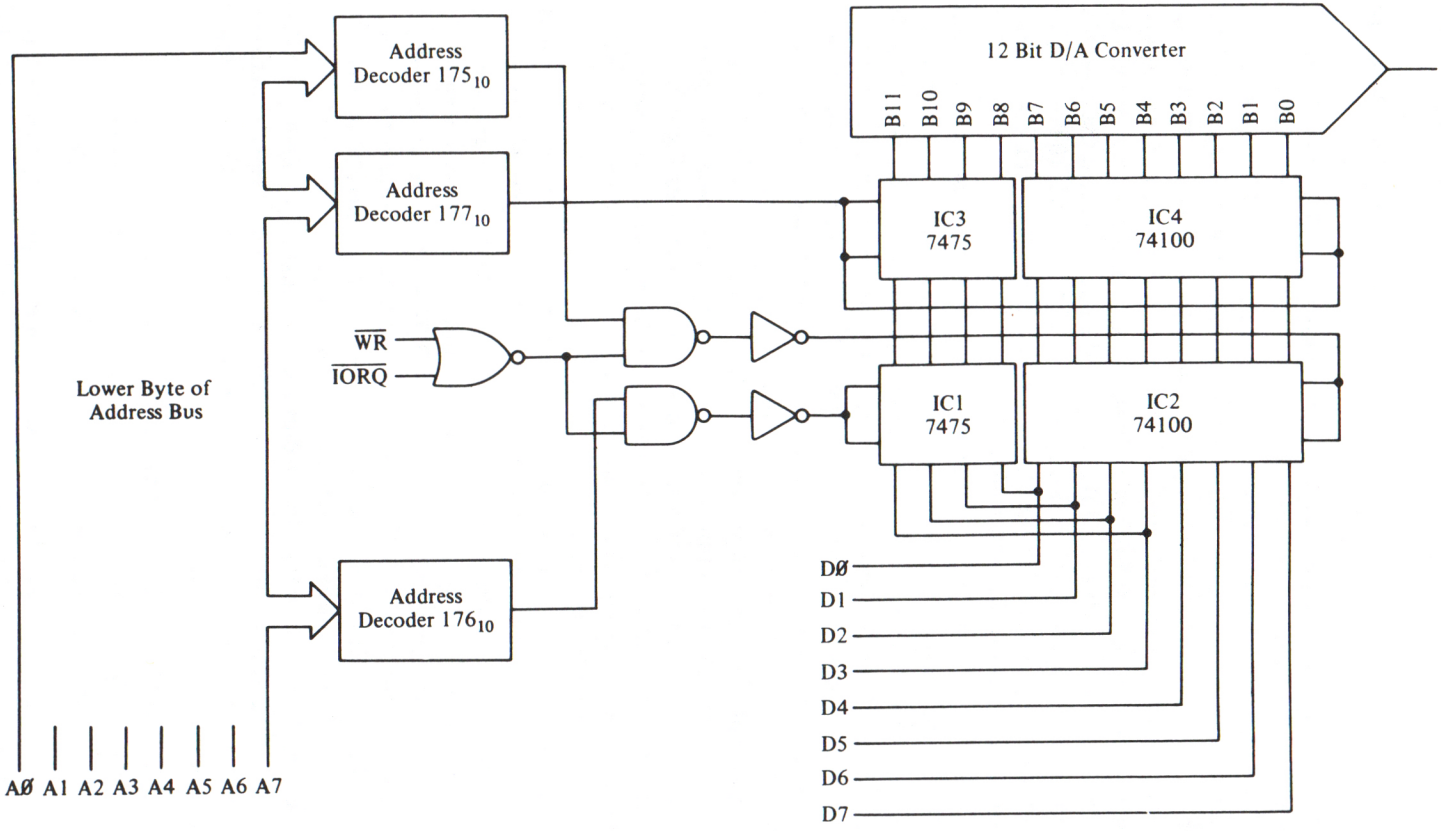


FIGURE 12-14. Interfacing eight-bit data bus to longer length DAC.

data in IC1 and IC2 into IC3 and IC4, respectively. To do that job we need software that does the following:

1. Loads the CPU accumulator with the lower 8 bits of the 10-, 12- or 16-bit data word.
2. Outputs the lower 8 bits to port 175_{10} (i.e., loads IC2).
3. Loads the CPU accumulator with the higher 2, 4 or 8 bits of the data word.
4. Outputs the contents of the accumulator to port 176_{10} (i.e., loads IC1).
5. Generates a dummy output instruction to port 177_{10} in order to generate a device select pulse to that port. This operation will transfer data from IC1/IC2 to IC3/IC4—and thence to the DAC digital inputs.

It is important that this double-buffered approach be used because any other approach will result in erroneous output to the DAC. In some cases, a single-buffered method is used because (a) the DAC contains its own internal data latches or (b) because the DAC has a chip enable line that can be turned on and off as needed.

DAC Output Circuitry

Many of the lower cost DACs used in electronic instrumentation are current output devices. This means that the analog output of the DAC will be an electrical current, not a voltage. Unfortunately, most of the display devices and other electronic circuits that will receive the DAC output are voltage-oriented. We can overcome that problem by converting the DAC output current to a proportional voltage. Figure 12-15 shows a simple method involving an operational amplifier operated as an inverter. The inverter input

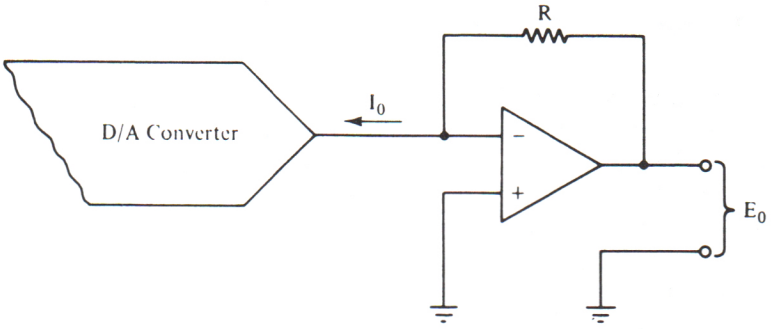


FIGURE 12-15. Creating voltage output for current output DACs.

is connected directly to the DAC current output and a feedback resistor scales the output voltage such that

$$E_0 = I_0 R$$

where E_0 is the amplifier output potential, in volts
I_0 is the DAC output current in milliamperes
R is the feedback resistor in kilohms.

If the output voltage polarity is incorrect, then there are several steps that can be taken to correct the problem. One is to use a DAC with complementary current outputs, and then connect the output to the operational amplifier that produces the polarity required. Another tactic is to follow the amplifier in Figure 12-15 with an additional inverter stage, keeping in mind that the amplitude will be the product of the gains of both stages and the output current. Another tactic will be to use a noninverting follower instead of the inverting follower in Figure 12-15. Of course, it is still necessary to make the current-to-voltage conversion, but that can be done with a simple resistor. Connect the DAC output to the noninverting input of the operational amplifier. From this same point to ground connect a precision resistor. The voltage applied to the noninverting input of the operational amplifier will be the product of the current and the resistance.

One last problem involves filtering of the output signal. The DAC output signal will contain steps equal to the 1-LSB value of the DAC. If we were to connect a binary counter to the digital inputs of the DAC, then the result would be a staircase version of the old-fashion sawtooth waveform. If we incorporate a low-pass filter between the DAC output and the display device, then this step-function is smoothed and will appear more like what we want to see. Select a filter cut-off frequency and slope that will smooth the signal without attenuating the main features of the waveform at the highest fundamental frequency of operation.

INTERFACING THE A/D CONVERTER

The A/D converter is the device that converts an analog input voltage or (sometimes) current into a proportional binary word that can be used in the computer. The subject of analog-to-digital conversion is beyond the scope of this chapter, but we will consider the A/D converter as a black box. Consider Figure 12-16. The typical A/D converter may have an analog input, a reference input (some use internal references), 8-, 10-, 12- or 16-bit binary digital output, and at least two service signals, *start* and *end of conversion* (EOC). Alternatively, some converters have a *status* line in lieu of the EOC line. The difference is that the EOC signal is a pulse that usually consumes but

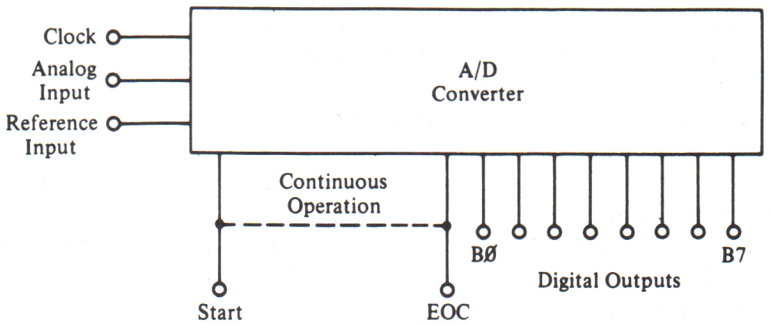


FIGURE 12-16. Basic A/D converter.

one cycle of the system clock (which may be either internal or external to the A/D), while the *status* signal will maintain one condition during the conversion process and a second condition when the data is valid. For example, one A/D converter has a *status* line that is HIGH when the conversion is taking place (which tells the computer and the outside world to not trust the output data) and LOW when the data is valid (indicating that the conversion is completed). The conversion time is the period from the issuance of a start pulse to the generation of the EOC pulse (or the status line equivalent). This is the fundamental limitation to the maximum speed of operation, hence the number of samples per second. Obviously, if it requires 100 mS to make a conversion, then it is unlikely that more than 10 samples per second are possible. The instrumentation designer must take into account the speed of the converter and the maximum sample rate required by the data being converted. If, for example, significant frequency components to 1000 Hz exist, then we must have (by Nyquist) at least 200 samples per second. This means a conversion not less often than every 0.5 millisecond, or 500 microseconds. In that case, a low cost 40 mS A/D converter simply would not do the job.

The A/D converter will remain dormant, doing nothing but marking time, until a start pulse is received on the *start* input line. It will then make the conversion and then issue an EOC signal. The A/D converter then goes into a dormant state once more to await the arrival of a start pulse. In some cases, this is the most desirable mode of operation. In other cases, however, we will want to make continuous conversions. The quickest way to do that is to connect the EOC line to the start line, and let the EOC signal be the start signal for the next cycle. In some cases, the EOC is coupled to the start line via an OR gate, so that power-on pulses or manual start pulses may also be accommodated.

As in most microprocessor interfacing chores, there are two basic approaches that can be taken: I/O-based and memory-mapped. In the I/O-based system we use the input and output instruction of the microprocessor (if

any), and that requires some form of existing I/O port. If we are using a pre-assembled computer for our design, then it may be possible to use existing I/O ports.

We may also have to interface directly to the bus lines inside of the computer or directly to the microprocessor IC. In any event, we will have to provide some means to recognize the microprocessor's control signals and decode the port address.

In memory-mapped systems, on the other hand, we will treat the output of the A/D converter as another memory location in the total memory bank. It is the usual practice to assign the A/D converter a location in the upper 32K of the 64K total memory that is common on 8-bit machines. That convention is probably the result of few computers, either personal or "instrumentation" applications, that have a full complement of memory. In fact, there will be a significant number of instrumentation applications that can be filled with 1K, 2K, 4K, 8K, and 16K memory sizes. There are partisans to both I/O and memory-mapped approaches, but the logic seems to be on the side of the memory-mapped proponents in cases where the computer has less than the full 64K of memory.

In Figure 12-17 we see an I/O-based scheme that works, even though it is wasteful of I/O ports. This method might be preferred, however, in a case where you are using a device such as the 6502 microprocessor with a 6522 versatile interface adapter (VIA) and only need a single input port. This system requires one complete input port on the computer to accommodate the A/D output data, plus one bit each of another input and an output port to accommodate the service signals from the A/D converter (*start* and *EOC*). In the example shown, we have (again) used port numbers 175 and 176. In this

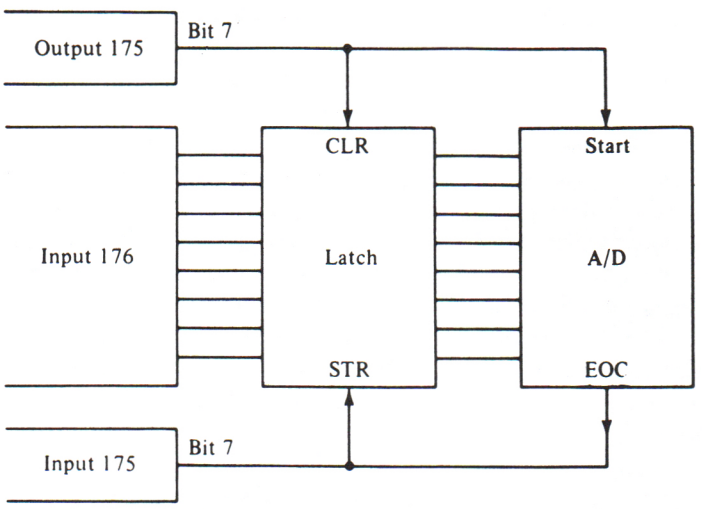


FIGURE 12-17. Connecting A/D converter to output port.

case, input port 176_{10} is used for the A/D output data, while bit no. 7 of input port 176_{10} is used to look for the EOC signal, and bit no. 7 of output port no. 175_{10} is used to generate the start signal. The A/D converter will initiate a conversion cycle whenever a HIGH appears on bit no. 7 of 175_{10}. The conversion will continue until complete, at which time a HIGH will be generated on the EOC line. This signal is applied to bit no. 7 of input port no. 175 (which tells the software that the data on 176_{10} input is good) and to the latch. In some cases, the latch will be activated by the EOC signal and then reset to LOW by the start signal.

A method that will accomplish the same job with less waste of I/O ports is shown in Figure 12-18. In this example we are multiplexing the port so that it can be used for both service and data input functions. In this case, we use a single I/O port, no. 175_{10}. In order to initiate a conversion, we will output on port 175_{10} any binary word that has bit-6 HIGH and bit-7 LOW; e.g., Ø1ØØØØØØ_2 (40_{16}). If the A/D converter is edge-triggered, or if latch no. 1 is actually a Type-D flip-flop, then we can leave B6 HIGH. Otherwise, we will

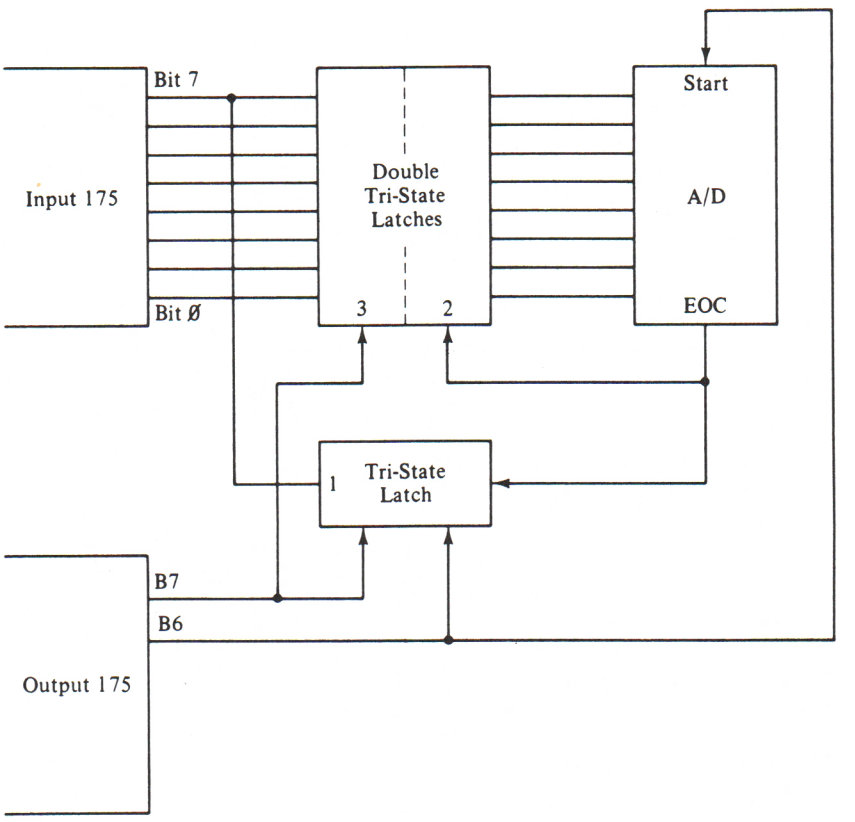


FIGURE 12-18. More efficient system.

follow this operation with another write to 175_{10} with ØØØØØØØØ. In this situation we have the A/D converter making the conversion, and latch no. 1 looking for the EOC pulse. Note that the output of latch no. 1 is connected to one bit of input port 175_{10}. The computer software will loop looking for bit-7 HIGH which indicates the A/D conversion is completed. The EOC pulse also cranks the output data into latch no. 2. After the program senses that the conversion is finished, it will output a HIGH on bit-7 of output port 175_{10}. This signal has the effect of turning on latch no. 3 (inputting the data) and turning off latch no. 1. In some cases, we might want to use separate bits for (first) turning off no. 1 and then turning on no. 3.

Figure 12-19 shows a simple method for connecting the A/D converter directly to the data bus. In this case, we are using the Z80 control signals for the example. For the example an I/O-based format is assumed. We could, however, make this a memory-mapped system through the simple expedient

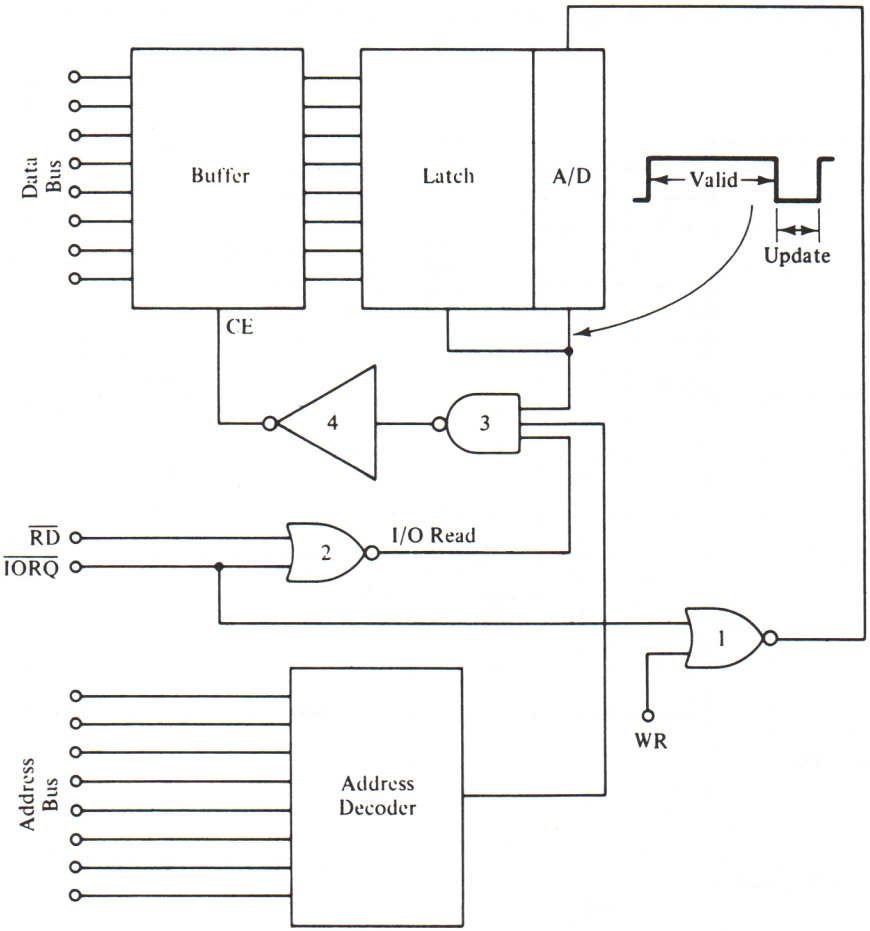


FIGURE 12-19. Interfacing A/D to data bus.

of using the memory request ($\overline{MREQ}$) signal instead of the Input Output Request ($\overline{IORQ}$) signal, as shown. The A/D converter selected (or designed) will have latched output lines (to keep the data valid as long as needed for the computer to respond) and a status line to indicate when the data is valid.

In the Z80 system, the OUT operation requires two signals: $\overline{IORQ}$ and the *write* ($\overline{WR}$); these indicate an *I/O write* which is an output. Similarly, an input operation requires $\overline{IORQ}$ and $\overline{RD}$; these form the *I/O read* signal. Gate no. 1 forms the start circuitry for the A/D converter. Since both of the control signals ($\overline{IORQ}$ and $\overline{WR}$) are active-LOW, we will use a NOR gate as an NAND-negative-logic gate. In this case, both inputs have to be LOW for the output to be HIGH. When the computer program executes a proper output command, then the output of gate no. 1 will go HIGH, initiating a start of conversion for the A/D. Immediately thereafter, the computer will loop, looking for a valid input. Gate no. 2 creates the I/O read, while gate no. 3 prevents it from occurring until the data is valid. During the conversion period the status line will be LOW, so gate no. 3 will not produce the required LOW output.

None of the A/D converter scenarios presented thus far are particularly appealing; all suffer from the necessity of tying up the computer looking for some input condition. In Figure 12-20 we see a system that takes advantage of the interrupt capability of the machine. A computer will respond to the interupt by temporarily putting aside the program being executed and switching to a subroutine that will service the interrupt. Let's take an instrumentation example of how this might be used. Assume that we are going to build a data logger and average for, say, an electroencephalograph (EEG) evoked potentials computer. The A/D converter will be continuously making conversions of the patient's brain wave signal. In the case of Figure 12-20, the converter is connected in the continuous, or asynchronous mode. We will input a data point, store in in memory, and then average it with the number of data points already received. Typically, we could add the "to be averaged" number in a 16-bit register pair (or two sequential memory locations), and keep a software (or hardware in some computers) count of the total number of samples. For a given time, the value in the averager will accumulate. When the "run" is completed, then the total accumulated value will be divided by the value of the counter. Of course, for a real evoked potentials computer, we would need a separate "averager" location for each data point.[1] The purpose of using the interrupt in this system is to allow the computer time to do the averaging and other chores while the data converter is making the conversion. When a new conversion is ready, the A/D circuit will interrupt the computer, input the new data, and then go on to make a newer yet conversion. During the time between the input operations the computer can do its "number crunching thing."

The start signal is coupled through a two-input OR gate. This

[1]See *Introduction to Biomedical Equipment Technology* by J. J. Carr and J. M. Brown, New York: John Wiley & Sons, 1981.

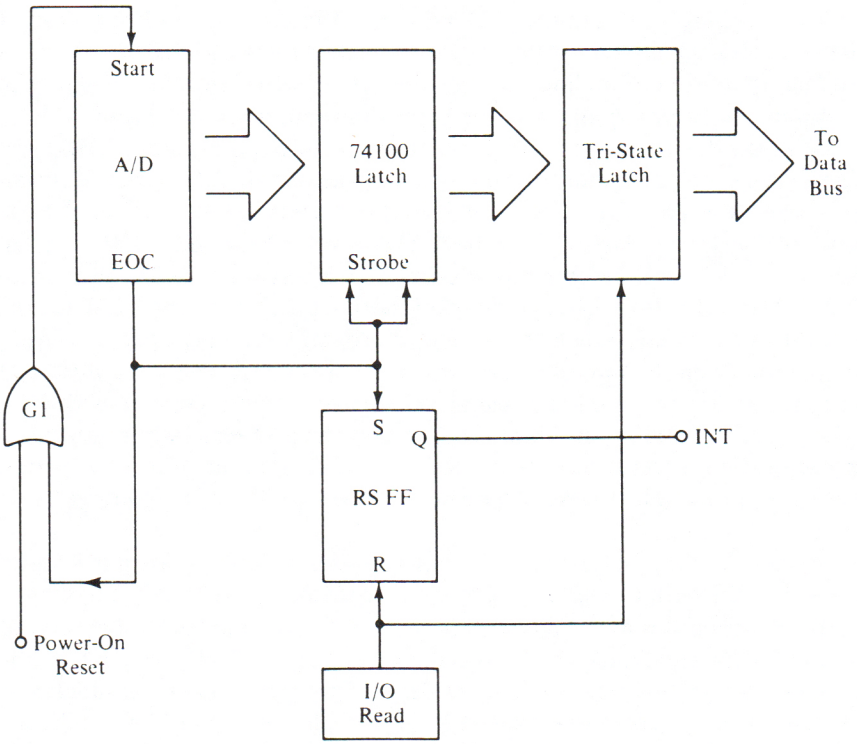


FIGURE 12-20. A/D driving interrupt line.

arrangement allows us to couple either the *EOC* signal or the *power-on reset* signal to the start terminal of the A/D converter. When the conversion is completed the EOC pulse is generated. This pulse will (a) input the A/D data to the 74100 latch, and (b) set the RS flip-flop (making Q = HIGH). If the computer wants to see an INT signal, then use the Q output of the RS FF. Otherwise, if an $\overline{\text{INT}}$ is needed (most often the case), then use the $\overline{\text{Q}}$ output of the RS FF.

When the computer finishes the execution of the instruction during which the INT or $\overline{\text{INT}}$ signal is received, it will store the location of the program counter on a stack some place in memory, and then switch to executing the interrupt subroutine. In the case of the A/D converter interrupt, that program will generate an I/O read operation. This means that a device select pulse for the I/O port servicing the A/D will be generated, so it may be used to (a) reset the RS FF, and (b) turn on the tri-state latch that places the A/D data (now on the output of the 74100 latch) onto the data bus of the computer.

As an alternative, we can connect the start line of the A/D to a second I/O read signal so that the conversion will start only under software control.

CHAPTER 13
CONTROLLING EXTERNAL DEVICES AND CIRCUITS

There are many applications for any microcomputer, including the T/S-1000, which depend upon controlling some external device. In Chapter 12 we learned how the T/S-1000 can receive data from its environment, while here we are going to discuss what the T/S-1000 can do to control external devices. There are several different schema that could be used, but we are going to assume that the user has purchased or provided as part of the external interface circuit an eight-bit I/O port. Most of the discussion will involve only single-bit control, so either parallel or serial I/O ports can be used for those applications. In other cases, however, we must use either a parallel output port, or, a serial system in which the external device contains a UART or some other means of converting the serial data stream from the computer into parallel bits.

We will not discuss peripherals such as printers in this section because they were discussed separately in Chapter 9. Here we will discuss items such as relays and optocouplers that can control AC-line powered devices, yet remain isolated from the computer. We will also discuss interfacing with lamps and LEDs, as well as numerical readouts. Keep in mind when reading the display section, however, that the T/S-1000 is already equipped with a video driver, so it is usually more efficient to output alphanumerical data to a TV screen. There are, however, a limited number of users who will use the T/S-1000 as a

small controller built into some larger system as if it were a mere component. Those readers may well have to understand how to interface LEDs and seven-segment displays.

ISOLATED CIRCUITS

When interfacing with high voltage (e. g., 115 VAC power line) circuits, it is not prudent to simply connect the T/S-1000 to the circuit and hope for the best. The semiconductors inside of the T/S-1000 will immediately become the victim of a silicon-to-carbon converter (i. e., blow out, rapidly) if subjected to such burdensome potentials. When it becomes necessary to control such potentials, then it will be wise to use some form of isolation.

Why would one want to interface with such circuits? If we want to be able to control heaters, furnaces, electrical lights, machinery and other devices, then it will be absolutely essential that we be able to safely interface the T/S-1000 computer to a high voltage circuit. Industrial users are faced with even more acute problems.

There are basically two forms of isolation readily available: electromechanical relay and optoisolator. There are also other methods, but these two are the most common.

Figure 13-1 shows how an electromechanical relay can be interfaced to the output port of a microcomputer, T/S-1000 included. Only one bit of the output port is used, so either serial or parallel forms are permissible. In this case, the interface device is a TTL open-collector inverter (U1) in which the relay coil (K1) is connected between its output and V+. Depending upon which TTL open-collector inverter is used, V+ can be anything from +5 VDC max. to +30 VDC max. When the output of U1 is HIGH, then the relay sees the same potential at both ends of its coil, so remains unenergized. If, on the other hand, the output of U1 is LOW, then one end of K1 is grounded and the other end is at V+, so a current flows in the K1 coil (the relay is energized, so the contacts close). Thus, since U1 is an inverter, a LOW at the output port (B7 in this case) will cause the relay to remain unenergized, while a HIGH will cause the relay to turn on. In this case, we can turn on the relay simply by writing any number to the output port which results in bit B7 being HIGH, so 8ØH fills the bill (this number is the same as $1ØØØØØØØ_2$ in binary).

The actual control of the high voltage circuit is through the relay contacts. In this drawing (Fig. 13-1) we have show SPDT contacts, although all combinations are possible.

There is one little limitation on this method however, and that is the current capability of the inverter output. The relay coil cannot draw more current than the inverter output stage can handle, or there will be trouble. Usually, this current is limited to 30 to 40 milliamperes, depending upon the

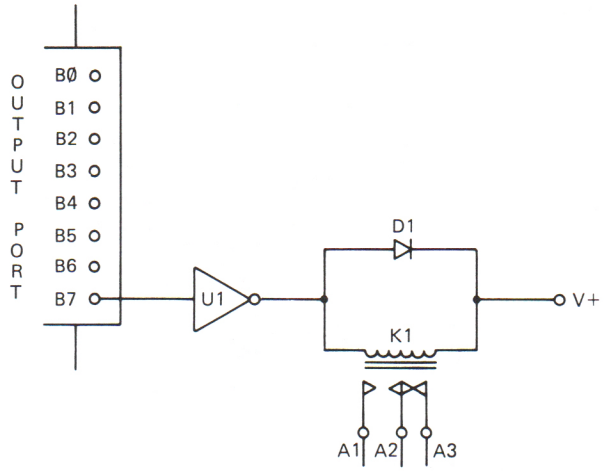


FIGURE 13-1. Driving a relay from a computer output port using TTL inverter.

style of inverter. There is a way around the problem, but more of that in a moment.

The diode (D1) used in Fig. 13-1 is supposed to suppress the "inductive-kick" spike created whenever the relay coil is deenergized. The coil of the relay is actually an inductor, and may have a right hefty inductance in some cases. When the magnetic field surrounding that coil during energization collapses, then the counterelectromotive force generated can create a tremendous (hundreds of volts) spike across the coil. The purpose of diode D1 is to suppress that spike. Normally, the diode will be reverse biased to ordinary DC operation, so is effectively out of the circuit. When the current in the coil collapses, however, the spike is of reverse polarity than the DC, so will forward bias the diode. The spike is clipped off to a value of 0.7 volts (the diode junction potential). For most applications (except the heaviest relays), a 1N4007 diode is sufficient for D1. This diode has a rating of 1000 volts PIV at a current of 1 ampere.

Heavier loads cannot be handled with the simple TTL inverter scheme shown in Figure 13-1. The current limitation of those inverters is only about 30-40 mA, and that is too small for many, perhaps most, relays and other forms of load. Figure 13-2 shows another method that uses an external transistor driver for the relay (or other- load). The transistor (Q1) must have a collector current rating sufficient to handle the current expected from the load, with a certain margin for safety. We must also pay attention to the collector voltage rating and to the collector power dissipation. The collector voltage ($V_{c\text{-}e(max)}$) must be sufficiently high to handle V+ (again with a safety margin), and the collector power dissipation (P_d) must be high enough to

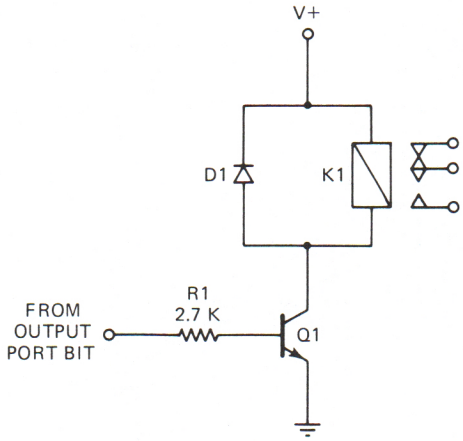


FIGURE 13-2. Transistor relay driver.

sustain the power developed. Normally, this power is the product of the relay coil current and the V+ voltage:

$$P = (V+)(I_{K1}) \tag{13-1}$$

where: P is the power in watts

$V+$ is the supply potential, in volts

I_{K1} is the relay coil current (or other load current), in amperes

It is not sufficient to merely multiply together the maximum collector current and collector voltage ratings of the transistor to find out whether or not this power is less than the rated permissible power. It is frequently the case that the product of the maximum collector current and the maximum collector voltage exceeds the maximum power dissipation. In other words, we may operate near maximum current, or, near maximum voltage, but *not at the same time!*

The transistor used in this case is an NPN type, so will turn on when the base is made more positive than the emitter. In this circuit, therefore, a HIGH applied to the input end of resistor R1 (marked "From Output Port Bit") will turn the transistor on. The value of R1 was selected assuming a moderate *beta* (i.e., gain) transistor and a TTL-compatible output port bit. When the input is LOW, then the transistor is turned off, so the relay is not energized. When the input is HIGH, however, a current flows in R1 to bias Q1, and thereby turns transistor Q1 on hard. The transistor is essentially saturated, so the collector is at a potential very close to ground. The relay is, therefore, grounded, so will turn on.

The diode in Fig. 13-2 serves exactly the same function as in Fig. 13-1. If the relay is a very heavy type, however, we must either connect two or more 1N4007 in series (to increase the PIV), or, buy another diode.

The beta of Q1 is also important. Most computer output ports will have a limited drive capability. On some devices, the available current might be 3.2 milliamperes (fan-out of 2), while on others it might be substantially more. Most normal TTL devices have a fan-out of 10, so will produce an output current of 18 mA. In order to select a proper transistor, we must know the available output current and the load current, I_{K1}. The minimum *beta*gain of Q1 must be:

$$H_{fe} = I_{K1} / I_b \tag{13-2}$$

where: H_{fe} is the minimum *beta* gain of Q1

I_{K1} is the collector load current expected (maximum)

I_b is the current available from the output port bit to drive Q1 base

After you calculate the minimum beta, multiply it by 1.25 in order to allow for variation between transistors and aging.

There will be no problem if you select a transistor with a higher *beta* than the minimum as calculated above, but a lower *beta* can result in erratic or nonoperation of the circuit.

There are two cases where we might want to use the circuit of Fig. 13-3.

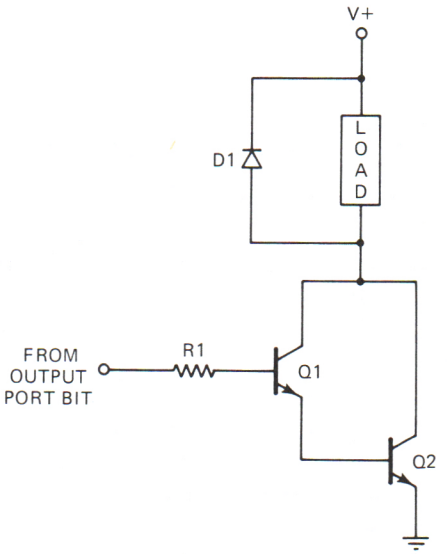


FIGURE 13-3. High gain relay driver.

First, where very heavy loads are used, e.g., a solenoid coil or heater element instead of a low-current relay. Second, where the load is moderate but the drive capability of the computer output port is limited. Almost all computer ports will deliver at least 3.2 mA (i.e., fan-out of 2), but some are limited to a fan-out of 1 (i.e., 1.8 mA). Under that circumstance, it requires either a TTL inverter with a higher drive capability, or, a circuit such as Fig. 13-3.

Figure 13-3 shows two transistors connected into a superbeta configuration called the Darlington amplifier, or, Darlington pair. Notice that one transistor (Q1) operates as an emitter follower to drive the base of the second transistor (Q2), and that both collectors are tied together. The overall beta of this transistor-pair, when we treat it as a single transistor, is the product of the individual *beta:*

$$H_{fe(t)} = H_{fe(Q1)} \times H_{fe(Q2)} \tag{13-3}$$

or, if Q1 and Q2 are identical and have the same *beta* gains, the total $H_{fe(t)}$ is the square of the common beta specification. Let's see what that does to the beta. Suppose we have two transistors each with a *beta* of 100. The *beta* of the Darlington pair made from these two transistors, then, will be $(100)^2$, or, 10,000. What this tells us is that the base current case can be $\frac{1}{10000}$ of the collector current. Let's suppose for a moment that we have one of those microprocessors or accessory chips which allows a maximum fan-out of 2. From those outputs we will obtain a maximum drive current of 3.2 mA. If the Darlington pair has a *beta* of 10,000, then, we find that the total permissible collector current will be (based on Eq. 13-2):

$$I_{K1} = (I_b)(H_{fe})$$

$$I_{K1} = (3.2 \text{ mA})(10{,}000)$$

$$I_{K1} = 32{,}000 \text{ mA} = 32 \text{ amperes} \tag{13-4}$$

Of course, we are assuming that the Darlington pair can handle 32 amperes, and the power dissipation that passing 32 amperes entails.

There are a couple of different alternatives to obtaining the Darlington pair. We can, and often do, use discrete transistors for this service. Where high current capability is needed, we often find a 2N3053 used as Q1 and a 2N3055 used as Q2. Or, alternatively, we can elect to use a ready-made Darlington transistor. Lambda Semiconductor and Motorola both manufacture Darlington pairs packages inside of diamond-shaped TO-3 transistor packages. These transistors have the appearance of being ordinary TO-3 power transistors, but are in fact Darlington pairs and exhibit superbeta characteristics.

Another method for isolating the computer from disastrous loads is the use of the optoisolator. These devices are integrated circuits in which a light emitting diode (LED) is positioned such that its light falls on the sensitive

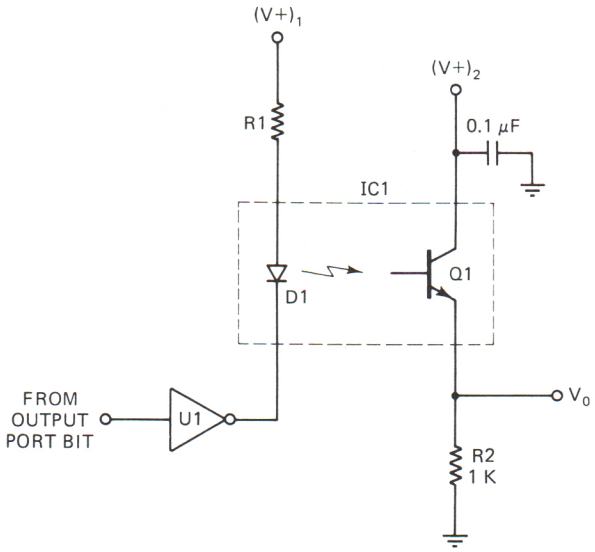


FIGURE 13-4. Optoisolator interfacing.

base region of a phototransistor. Figure 13-4 provides an example. The transistor acts like a switch in this case. When the LED is turned on, and its light falls on the transistor base, the transistor is forward biased. It will conduct heavily, thereby creating an output potential V_0 across the resistor. If the LED is turned off, however, the phototransistor will not conduct, and there will be no output voltage.

The LED can be turned on and off through any of several means. Here we show a TTL open-collector inverter that uses the LED and its current-limiting resistor (R1) as the load. When the input of the inverter is LOW, the output is HIGH, so the LED remains off. But, when the input of the inverter is HIGH, the inverter output is LOW, so the LED is grounded and will therefore turn on. The input of the inverter can be controlled from one bit of a computer output port.

The optoisolator can be used alone, or, to control relays and so forth. In many cases, it will be used to supply the gate turn-on current required by a silicon controlled rectifier (SCR) or other Triac device. Those devices are quite handy in controlling AC loads, and form the basis for those light dimmer switches one can install in homes in place of ordinary switches.

DISPLAYS

Displays tell the operator of some computer or computer-based instrument just what is the status of the system, or, will communicate some bit of

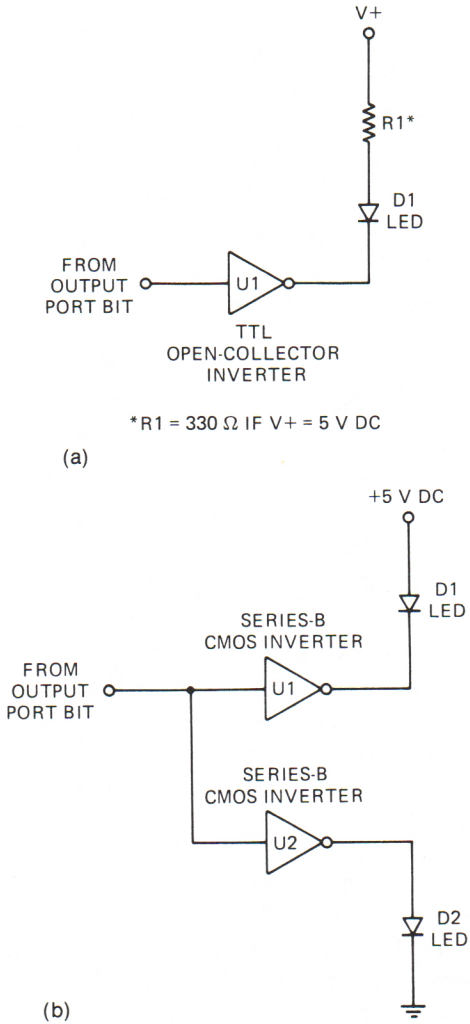


FIGURE 13-5 (a) Interfacing LED to TTL open-collector inverter. (b) Interfacing LED to CMOS inverters.

information—like where the fire is located in the building (in the case of a computerized annunciater panel!). There are several different options for display technology open to us on the T/S-1000. Perhaps the easiest to use is the TV monitor that one ordinarily uses with the computer. We might also want to use a light emitting diode (or several, possibly of differing colors). Or, we may want to use a seven-segment LED numerical display to output a number or some funny symbol to the user. In this section, we will discuss the possible ways these jobs can be done.

The light emitting diode (LED) can be interfaced in the same manner as the LED in the optoisolator of the previous section. Figure 13-5(a) shows the proper method for connecting an LED to a computer output port bit. The inverter is a TTL open-collector type on which the load is an LED with its current-limiting resistor. If the V+ supply to the LED is +5 volts, then we can use a resistance of 330 ohms for R1, but if higher, scale the resistance up sufficiently to keep the LED current within tolerance (usually 15 mA on most LEDs). The description of how this circuit works was given above, so will not be repeated here.

Figure 13-5(b) shows how we can interface 15 mA LEDs to a computer through CMOS inverters of the series-B type (series-A lacks the current sinking capability). There are two versions shown here. Inverter U1 connects the LED between the + 5 volt power supply and the inverter output (note the lack of a resistor). When the input of this inverter, i.e. the computer output port bit, is HIGH, then the inverter output is LOW, so the cathode of the LED is grounded—the LED turns on. When the input of U1 is LOW, however, the output is HIGH so the cathode of the LED is ungrounded... and the LED is turned off.

Inverter U2 in Fig. 13-5(b) is configured oppositely from inverter U1. In this circuit the LED is connected from the inverter output to ground. When the inverter input is HIGH, then its output is LOW so no potential appears across D2—the LED is off. If, on the other hand, the input of U2 is LOW, then the output is HIGH so a potential is applied to D2—the LED is turned on.

Seven-segment LED displays use seven bar-type LEDs to represent the ten digits of the decimal numbers system. We can inteface these devices in exactly the same manner as in Figs. 13-5(a) and 13-5(b), except that more output port bits will be required (seven, in fact). The programmer will then conspire to output the proper seven-segment code when it is time to display any given number.

Incandescent lamps generally require more current than even the largest LEDs. Although some will fall into the 30-40 mA category permitted by some open-collector TTL inverters, most do not. Currents in the 150 to 1000 mA range are not uncommon. For those, use a transistor circuit such as Fig. 13-2, but with the diode removed and the lamp used as the load.

PART III
MACHINE LANGUAGE PROGRAMMING

CHAPTER 14
Z80 ADDRESSING MODES

The many different instructions offered by the Z80 microprocessor reflect, in part, a number of different addressing modes for the same basic operations. In all cases, the Z80 instructions pertain to operations on data between internal registers, in the external RAM or ROM memory, or input and output ports. It is, perhaps, easier to realize just where such a large, magnificient instruction set comes from when you consider that there are several different forms of operation, on data in up to eighteen 8-bit registers, and four 16-bit registers. And since the Z80 uses a 16-bit address bus, it can accommodate a mix of *random access memory* (RAM) and *read only memory* (ROM) up to 2^{16} (65,536, or so-called, "64K") one-byte locations. There are also up to 256 input and 256 output port selections possible.

One of the advantages of the Z80 is the large number of addressing modes, taking it out of the simple process controller stage and making it a real live computer. The modes of addressing offered by the Z80 device include:

Immediate addressing
Immediate extended addressing
Modified page-zero addressing
Relative addressing
Extended addressing
Indexed addressing

Register addressing
Implied register addressing
Register indirect addressing
Bit addressing

In this chapter we will discuss the various different types of addressing, and then deal with sample instructions (enumerated more fully in Chapter 19), using examples.

Immediate Addressing. In the immediate addressing mode, the operand follows the op-code in sequential locations, and the operand is loaded into the selected location immediately. A prime example of the immediate addressing mode is the ADD A,n and SUB A,n instructions. In these instructions, the operand n in added (or subtracted if SUB A,n) to the contents of the accumulator, and the result is then stored in the accumulator.

The format for the immediate addressing type of instruction is shown below:

byte 1	op-code
byte 2	(n)

Example

The op-code for the ADD A,n instruction is 11000110 in binary or C6 in hexadecimal. Let us say that the accumulator contains A7 (hex) before the following code is encountered:

memory location	code	
06 00	C6	byte 1
06 01	Ø7	byte 2

This means that the instruction fetched (C6) is the ADD A,n instruction, and that operand n (the next sequential location) is Ø7 (hex). After the execution of this two-byte instruction, therefore, the contents of the accumulator will be

A → A + n
A → A7 + Ø7
A contains AE

The main utility of the immediate addressing mode is to load data into specific registers or locations or to perform arithmetic operations using constants.

Immediate Extended Addressing. This form of addressing is merely immediate addressing extended so as to accommodate 16-bit data transfers. While the immediate addressing type of instruction can be only two bytes (e.g., the op-code and operand n), the extended immediate type of instruction requires three bytes of data (op-code and two following n bytes). The format of this type of instruction is

```
byte 1   op code
byte 2   (n1)
byte 3   (n2)
```

An example is the load HL instruction that causes two operands to be loaded into the 16-bit HL register pair. LD HL,nn would look like:

```
byte 1   21   op-code for LD HL,nn
byte 2   n    low-order byte for HL
byte 3   n    high-order byte for HL
```

There are also similar instructions for the other 16-bit register pairs, namely, LD, BC, nn; LD DE,nn; and LD SP,nn.

Example

Suppose we wanted to load the HL register pair with the 16-bit binary word 00111101 01101111 (3D 6F in hex). The program would look like

```
byte 1   21   op-code for LD HL,nn
byte 2   6F   data for low-order byte of HL
byte 3   3D   data for high-order byte of HL
```

Following the execution of this instruction, the the low-order (L) side of the HL register pair would contain 6F (hex) and the high-order side (H) would contain 3D. Taken together, these data form the 16-bit word 3D6F (hex).

Modified Page-Zero Addressing. This type of instruction allows the programmer to call any of eight memory locations on page zero (i.e., first 256 addresses, starting at ØØ hex). The example of this type of addressing is the RST p instruction which, depending upon operand p, will reset the program counter to any one of the following addresses on page zero of memory: ØØØØ, ØØØ8, ØØ1Ø, ØØ18, ØØ2Ø, ØØ28, ØØ3Ø, ØØ38. In this instruction, the current contents of the program counter are pushed onto an external memory stack. The high-order byte of the program counter is loaded with ØØ (hex), while the

low-order byte is loaded with a word that selects any of the eight locations called out above. For example, loading the low-order byte of the program counter with CF (hex) would cause the instruction to be RST,Ø8.

The main use of the modified page-zero addressing is to allow servicing of subroutines with a single-byte call instruction. Consider Fig. 14-1. We are executing a program in page 1Ø, and at location 1ØB2 encounter a CF during an instruction fetch cycle. This tells the Z80 that it is a modified page-zero call to a subroutine at location ØØ Ø8, so program control jumps to location ØØ Ø8. Note that the program counter (PC) contained 1Ø B2 when the RST, Ø8 instruction was encountered. The high-order byte of the PC is pushed into stack pointer location SP-1, while the low-order byte is in location SP-2.

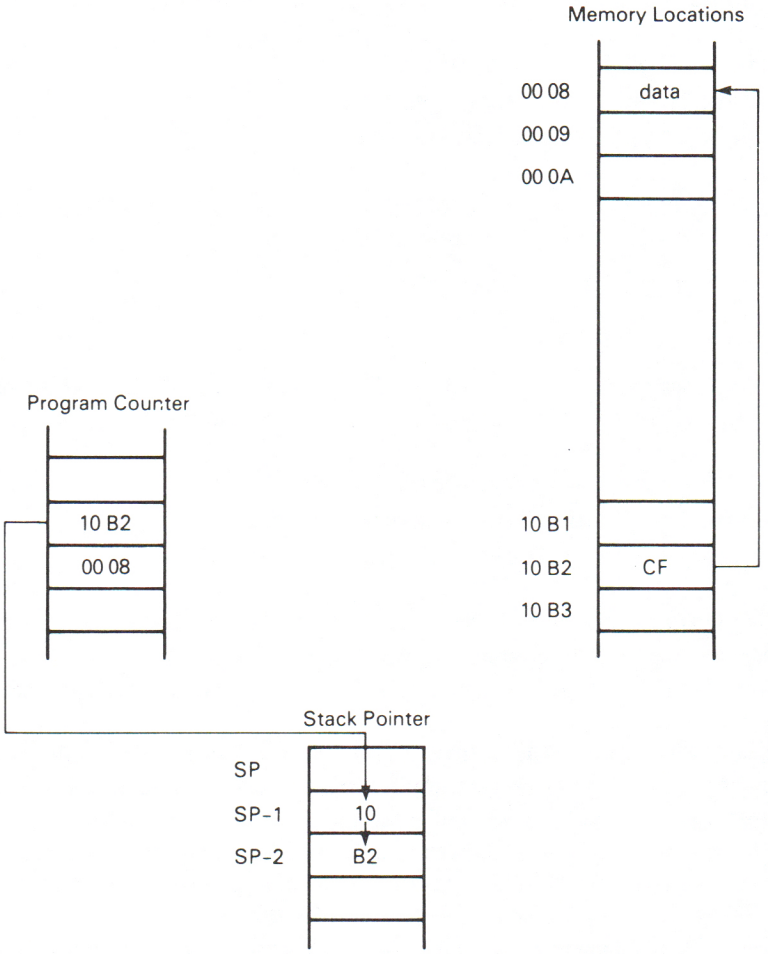


FIGURE 14-1.

When program control returns from the subroutine, then the program counter would reclaim the data in the stack, and increment appropriately to pick up the next instruction at 1Ø B3.

Relative Addressing. The relative addressing mode allows jump instructions in two bytes that cause program control to shift by a displacement integer *e*. The jump will occur to a memory location that is −126 to +129 locations away from the current address. These instructions are two-byte instructions, so the value of *e* would be -128_{10} to $+127_{10}$. The displacement integer *e* is always represented as a signed two's complement number, so these values in binary would be 10000000 to 01111111 (8Ø to 7F in hex). Since this is a two-byte instruction, and the jump cannot occur until the instruction is finished, the program counter will increment *twice* before the jump occurs. This accounts for the difference in the two ranges of decimal numbers given above. The format for this type of instruction is

byte 1 op-code
byte 2 displacement integer (-128_{10} to $+127_{10}$)

Figure 14-2 shows a typical example, using the unconditional jump instruction JR,e. This instruction will cause an unconditional branch to a subroutine located at a displacement e from the op-code. The value of the second byte will be a e−2. Note that we encounter the op-code 18 (hex) at location ØØ Ø9 and displacement integer FA (two's complement for −6) at location ØØ ØA. After the execution of this instruction, the program counter will contain the new address, ØØ Ø4 (hex).

LOCATION	DATA	
ØØ ØØ		
ØØ Ø1		
ØØ Ø2		
ØØ Ø3		
ØØ Ø4	PC after execution	
ØØ Ø5		
ØØ Ø6		
ØØ Ø7		
ØØ Ø8		
ØØ Ø9	18	op-code for JR,e
ØØ ØA	FA	2's complement for − 6
ØØ ØB		
(etc.)		

FIGURE 14-2.

Extended Addressing. In extended addressing, we are allowed to use two 8-bit integers nn to create a 16-bit address. In the typical extended addressing scheme, there will be a one- or two-byte op-code, followed by two address bytes or operands. In either case, the first n byte is the lower-order byte, while the second is high-order byte. The format would be

byte 1	op-code
byte 2	(possible additional op-code)
byte 3	n1
byte 4	n2

Example

LD A, (nn) is an extended addressing instruction that tells us to load the accumulator (i.e., A register) with the byte located at a memory location given by two-byte operand nn. Note that the use of parentheses around the operand tells us that we mean the "contents of location nn," rather than the value nn. The code for this is

byte 1	3A	op-code for Ld A, (nn)
byte 2	n1	low-order byte of address
byte 3	n2	high-order byte of address

Indexed Addressing. This type of addressing uses the two 16-bit index registers (IX and IY), plus a displacement value following the op-code, to compute the effective address of a jump. In a typical indexed addressing instruction, there will be a two-byte op-code, followed by the displacement integer d. The format of indexed addressing instruction is given below.

byte 1	(op-code)
byte 2	(op-code)
byte 3	(d)

Consider, for example, the LD (IX + d), A instruction. This instruction causes the memory location pointed to by the contents of the IX index register and displacement integer d to be loaded with the contents of the accumulator. The code would be as follows.

byte 1	DD	(op-code for LD (IX + d), A)
byte 2	77	(op-code for LD (IX + d), A)
byte 3	d	displacement integer d

Example

The accumulator contains 3F (hex), and the IX index register contains 44 00 (hex). The code DD 77 05 is encountered on an instruction fetch. This indicates a LD (IX + d), A instruction, in which d is 05 (hex). The memory location 44 00 + 05 = 44 05 will contain 3F (hex) after the execution of this instruction.

Register Addressing. This addressing mode allows us to transfer data between different registers of the Z80. An example is the LD r,r′ instruction. Registers, A, B, C, D, E, H, and L can be used of either r or r′. We make up the one-byte op-code using the register codes in appropriate spots in the op-code, using the format as shown below:

0 1 | ← r → | ← r′ → |

Example

The register code for the D register is 010, and the register code for the E register is 011. Since the operation of the LD r,r′ instruction is r - - r′, if we want to load the D register with the contents of the E register, we would use the op-code 01010011.

Implied Addressing. In the implied addressing mode we use special instructions that always use the same CPU register to contain the operands. An example of this type of instruction is the LD R, A instruction, which will load the R (refresh memory) register with the contents of the accumulator.

Register Indirect Addressing. This powerful type of instruction causes the transfer of data between the CPU and a memory location pointed to by the contents of one of the 16-bit register pairs. An example of this type of instruction is the LD (DE),A. This mnemonic is read, "Load the memory location pointed to by the contents of the DE register pair with the contents of the accumulator."

Example

The accumulator contains 9D (hex), and the DE register pair contains 65 08 (hex). If the following is encountered,

00010010 (op-code for LD (DE), A)

then memory location 65 08 will contain the byte 9D (hex) after the execution of this instruction.

Bit Addressing. One of the principal advantages of the Z80 in many types of programming is the ability to set, reset, or test the condition of any single bit in any register. The op-code will be of the form

```
byte 1   (op-code)
byte 2   0 1 _ b _ _ r _
```

Example

The bit code (b) for bit 5 is 101, and the register code for the D register is 010. In order to test the condition of bit 5 in the D register, we would use the Bit 5,D instruction with the op-code

```
byte 1   1 1 0 0 1 0 1 1
byte 2   0 1 1 0 1 0 1 0
```

In this particular case, the Z flag in the F register would be SET if the tested bit is Ø and RESET if the bit is 1.

CHAPTER 15
THE Z80 INSTRUCTION SET (GENERAL)

There are 158 instructions that can be executed by the Z80. Actually, when they are broken down into their various forms, counting the various modes of instruction, we count over *400* Z80 instructions. In a previous chapter we classified these instructions according to the addressing mode used. But in this chapter, we will classify the instructions in their respective groups. The groups are

Load and exchange
Block transfer and search
Arithmetic and logical instructions
Rotate and shift
Bit manipulation
Jump, Call, and Return
Input/Output
CPU control

LOAD INSTRUCTIONS

The load instructions are used to move data from one location to another. More specifically, there are two basic types of LOAD instruction: (a) move

data from one internal register to another internal register, and (b) move data from CPU registers to/from external memory locations.

There are two basic groups of LOAD instructions: the 8-bit and the 16-bit LOAD instructions. These instructions move single bytes of data, and use the 8-bit registers and 8-bit memory locations (sometimes pointed to by 16-bit index registers).

The 16-bit LOAD instructions use the register pairs AF, BC, DE, HL, SP, IX, and IY. They also use two-byte memory addresses to specify the locations of two bytes of data to be moved into the CPU, or from the CPU.

In all of the LOAD instructions, there must be specified both a source of the data and a destination. These may be an internal register or a memory location.

EXCHANGE INSTRUCTIONS

The exchange instructions are used to exchange the contents of any two specified registers. These are the instructions that are represented by the mnemonics EX and EXX.

BLOCK TRANSFER AND BLOCK SEARCH

One of the nicest things about the Z80 is the block search and transfer instructions. The transfer instructions include LDI, LDIR, LDD, and LDDR. The search instructions are CPI, CPIR, CPD, and CPDR.

The block instructions use three 16-bit register pairs in their execution:

HL Address of source location
DE Address of destination location
BC Byte counter

In any program using these instructions, it is necessary for these registers (HL, DE, BC) to be initialized to the value required. When the block instructions are executed, these registers are automatically incremented to point to the next location.

The block transfer instructions are defined as follows:

LDI Load and increment. This instruction moves one byte of data from the location pointed to by the HL register pair to the location pointed to by the DE register pair. After the execution of an LDI instruction, the HL and DE registers are incremented (pointing to the next sequential location), and the BC register is decremented.

LDIR This is the same as the load and increment instruction class, but will *load-increment-repeat.* The same operation is repeated until the contents of BC are zero. If the BC register is initialized to a specified value, then this one instruction will be used to move an entire block of data from one location to another.

LDD
LDDR The LDD and LDDR instructions are analogous to LDI and LDIR instructions, respectively. The difference is that the HL and DE register pairs are decremented, instead of incremented, after each execution. The BC register decrements, as in the other group. These instructions transfer data from the highest location to the lowest, while the opposite takes place in the LDI and LDIR instructions.

The block search instructions are as follows:

CPI Compare and increment. This type of instruction will compare the contents of the accumulator with the contents of the memory location pointed to by the contents of the HL register pair. The result of the comparison is reflected by the condition of the flag register bits. After the execution of this instruction, the HL register pair contents are incremented, and the BC register pair contents of HL point to an address in memory, while BC is a byte counter.

CPIR In this type of instruction, a CPI operation is performed and repeated, until either of two conditions are found: (a) the byte counter (BC) is zero, or (b) the data in the addressed memory location matches the data in the accumulator.

CPD
CPDR These are the *compare and decrement* and *compare, decrement, and repeat* instructions. They are analogous to the CPI and CPIR instructions, respectively, but the HL register pair is decremented after execution, instead of incrementing.

ARITHMETIC AND LOGICAL INSTRUCTIONS

There are two groups of arithmetic and logical instructions: 8-bit and 16-bit. In the 8-bit arithmetic group are the *addition* (ADD), *add with carry* (ADC), *subtract* (SUB), and *subtract with carry* (SBC). The 8-bit logical group consists of AND, OR, or XOR instructions.

Also included in this 8-bit classification are *compare* (CP), *increment* (INC), and *decrement* (DEC) instructions.

The same basic set of ADD, ADC, SBC, INC, and DEC instructions are used in the 16-bit arithmetic/logical group, but instead of the 8-bit registers (A, B, C, D, E, H, and L), and single memory locations, they use the register pairs (HL, IX, IY, BC, DE, and SP).

There is also a series of instructions that decimal adjust the accumulator data (DAA). These can be used to allow multiprecision BCD numbers, signed or unsigned binary numbers, or two's complement signed numbers.

There are four additional instructions in the arithmetic/logic group: *complement accumulator* (CPL), *negate accumulator* (NEG), *complement carry flag* (CCF), and *set carry flag* (SCF). The CPL instruction causes the number in the accumulator to be complemented. This means that the 1's become 0's and the 0's become 1's. The NEG instruction causes the contents of the accumulator to be expressed in two's complement form. The CCF instruction causes the carry flag to be complemented. If the carry flag is 1, then it will become 0, and if it is 0 it will become 1. The SCF instruction will cause the carry flag to be set (i.e., made 1).

ROTATE AND SHIFT INSTRUCTIONS

The rotate and shift group instructions include RLC, RRC, RL, RR, SLA, SRA, SRL, RLD, RRD, RLCA, RRCA, RLA, and RRA. These instructions move specified bits left or right, according to rules given in Chapter 17.

BIT MANIPULATION INSTRUCTIONS

One of the things that make the Z80 one of the better uP chips is the existence of the bit manipulation instructions. We can test a bit for 1 or 0, we can reset a bit (RES), and we can set (SET) a bit. The particular bit tested, set, or reset can be in any specified register (A, B, C, D, E, H, and L), or any memory location. In the latter case, we may use either indexed or register indirect addressing of the selected memory location. There are quite a large number of individual instructions in this group, because we can select any of eight bits (0–7), seven different registers, or memory locations specified by HL, IX or IY register pairs.

JUMP, CALL, AND RETURN INSTRUCTIONS

A digital computer executes instructions in a sequential manner. In the ordinary course of events, the program counter is incremented one to several counts every time an instruction is executed. The number of counts

incremented is determined by the number of bytes required for the particular type of instruction. Although this sequential execution is one of the powerful aspects of digital computers, it would also limit the range of possible problems that could be solved to those amenable to direct sequential processing. It would be impossible to perform most operations requiring even the simplest decision. Even the simple matter of inputting data would become impossible. In those operations, an input port is connected to a keyboard. We create a loop, using a JUMP instruction that tests the strobe bit (usually B7), and if none is found, jumps back to the beginning of the loop. If a strobe is found, on the other hand, the program is allowed to fall through to the next instruction (usually an input instruction). But this is merely a trivial example. Most problems requiring decision logic on the part of the computer could not be performed without the use of the JUMP, CALL, and RETURN instructions. The instructions in this group include JP, JR, CALL, DJNZ, RET, and RETN.

A JUMP (JP mnemonic) instruction is a branch to a subroutine at some address other than the next address in sequence. The address at which the next instruction (i.e., the first instruction of the subroutine) is to be found is loaded into the program counter (PC). We may use either of three addressing modes: immediate extended, register indirect, and relative.

Each of the different types of JUMP instruction are keyed to certain conditions that are reflected by the status bits of the flag (F) register. The conditions that may be specified by the selection of the op-code include carry, noncarry, zero, nonzero, parity even, parity odd, sign negative, sign positive, and unconditional.

The conditional jump instructions look for the status of the appropriate bit of the flag register. If the condition is met, then the jump operation occurs.

In immediate extended addressing, the jump occurs to a 16-bit memory address specified by the two bytes following the jump instruction. If the condition called for is met, then the program control will be shifted to the location specified by the following two bytes.

Register indirect addressing allows us to store the 16-bit address of the first instruction in the subroutine we wish to execute in one of the three double registers. Either the HL, IX, or IY register pairs can be specified by appropriate selection of the op-code. The register indirect jump instructions are all unconditional.

A relative addressing JUMP (JR mnemonic) instruction exists for each of the following conditions: carry, noncarry, zero, nonzero, and unconditional. In this type of jump instruction the next instruction for the program to execute (i.e., the first instruction op-code in the subroutine) is specified by the current contents of the PC added to a displacement integer e. The value of e can be anything in the range −126 to +129, as measured from the address of

the instruction op-code (rather than the location of the displacement integer).

In the case of one of the unconditional jump instructions (all forms of jump used in the Z80 will recognize an unconditional "condition"), the program counter is loaded immediately with the two bytes immediately following the op-code for the jump. The second byte of these three-byte instructions becomes the low-order byte of the PC address, while the third byte of the instruction becomes the high-order byte of the PC address. Since the contents of the PC are now changed, program control is transferred to the instruction located at a memory location specified by the PC.

There is also one special form of jump instruction that is very useful: DJNZ. This stands for decrement register B and jump if it is nonzero. This instruction allows us to use the B register as a byte counter. We load register B with an integer equal to the number of times that we wish to execute a subroutine. The program control will transfer to the subroutine specified by a displacement integer e (this is relative addressing) as soon as the DJNZ is encountered. When the program control returns, the B register is decremented. If this operation does not bring the contents of B to zero, then the jump occurs again. This will continue until B counts down to zero. If B was preloaded with zero, then the program will jump and loop through all 256 bytes before terminating when zero is again encountered.

An example of a possible application of this type of instruction is in signal averaging. If we want to average 100 data points, then we can nest an input instruction inside a DJNZ loop. The B register is loaded with 100, and is decremented with each execution. When all 100 data points are input, then the B register decrements one more time to zero, terminating the operation. The program then falls through to the next instruction in sequence. Note that the relative displacement integer e is expressed in the form of a two's complement number.

The CALL instruction is a special case of the JUMP instruction. If the CALL is used, then the address of the memory location immediately following the CALL instruction is loaded into an external memory stack (pointed to by the SP register contents). This allows us to branch to a subroutine, and then return to the main program sequence.

The return (RET) instruction is a reverse call instruction, and is used to return to the main program once the subroutine jumped to by the CALL instruction is finished. The RET instruction is usually the last instruction in the subroutine. When this instruction is encountered, the program counter is loaded with the contents of the external memory stack (again, pointed to by SP). This will be the addresss of the first instruction following the CALL instruction that instigated the subroutine branch.

There are two specialized return instructions, RETI and RETN. These are for returning to main program control after the servicing of an interrupt and a nonmaskable interrupt, respectively.

INPUT/OUTPUT INSTRUCTIONS

Input/output instructions cause data to be input to, or output from the CPU. The Z80 uses several different registers, and has I/O instructions allowing direct use of these registers without first requiring the programmer to transfer the contents of the accumulator. In addition, there are several block I/O instructions.

Immediate addressing is available to the accumulator (i.e., register A), while the B, C, D, E, H, and L registers use register indirect addressing. The block I/O instructions are also register indirect.

Perhaps the most common I/O instructions are the immediate input and immediate output instructions (IN A, n and OUT A,n). In these instructions, the operand n is the eight-bit address of one of 256 (000–255) possible ports. This address will appear on the lower eight bits (A0–A7) of the address bus, while the contents of the accumulator appear on the upper eight bits of the address bus (A8–A15). The input or output data are passed over the eight-bit data bus to, or from, the accumulator, respectively.

In the case of register indirect addressing, the contents of the C register specify the eight-bit address of the selected port. This address (i.e., contents of C) are passed over the lower byte of the address bus to signal the device being selected. Also at this time, the contents of the B register are passed over the high-order byte of the address bus. This is analogous to the immediate I/O instructions, except that the sources of the data passed to the address bus are different.

The block input instructions include INI, INIR, IND, and INDR. The block output instructions include OUTI, OTIR, OUTD, and OTDR. They are analogous to the memory block transfer instructions, except that they use the contents of register pair HL to point to an eight-bit I/O address at an external memory location. In these instructions, register B is used as a byte counter. As in the register indirect case above, the contents of the C register hold the address of the I/O port. Note that this means that the contents of register pair HL point to a location in external memory. The contents of this location are then loaded into register C. When the actual transfer takes place, the contents of C are then placed on the low-order byte of the two-byte address bus. Also, as in other register indirect I/O instructions, the contents of the B register (i.e., byte counter) are placed on the high-order byte of the address bus.

CPU CONTROL INSTRUCTIONS

There are seven instructions in the Z80 set that are used exclusively for the control of the CPU: NOP, HALT, DI, EI, IM0, IM1, IM2.

The NOP instruction is a "no operation" instruction. During the execution of the NOP, the CPU will do absolutely nothing.

The HALT instruction causes the CPU to cease operations until an interrupt is received. The DI instruction disables the interrupts, while EI enables interrupts.

The IM0, IM1, and IM2 instructions allow the programmer to set any of three interrupt modes. The zero-mode (IM0) causes the Z80 to think that it is an 8080A. The IM1 causes program control to transfer automatically to location 00 38 (hex) when an interupt is received. Interrupt mode 2 (IM2) allows indirect call to an interrupt service subroutine at a location specified by two bytes: the contents of the I register and the 8-bit word received from the interrupting device. This feature allows vectored interrupts serving several peripheral devices whose subroutines may be different.

CHAPTER 16
SERVICING INTERRUPTS

Interrupts are a powerful tool on any programmable digital computer. The designers of the Z80 microprocessor chip, probably well aware of this fact, built into the device four ways to interrupt the CPU: nonmaskable, maskable mode-Ø, maskable mode-1, and maskable mode-2. In Chapter 8 we briefly discussed these interrupts, and then concerned ourselves with the hardware aspects of the Z80 interrupt system. In this chapter, we will expand the topic of interrupts by considering the programming aspects of servicing the interrupt request.

NONMASKABLE INTERRUPTS

The nonmaskable interrupt is always recognized by the CPU, regardless of the programming being executed. The nonmaskable interrupt goes into effect following the completion of the instruction currently being executed, and is initiated by bringing the $\overline{\text{NMI}}$ terminal of the Z80 (i.e., pin 17) LOW. This terminal is sampled by the CPU during the last clock pulse (i.e., T-period) of each machine cycle. If $\overline{\text{NMI}}$ is found to be LOW when this sample is taken, then the CPU will automatically begin the interrupt sequence on the next clock pulse.

One principal difference between the nonmaskable interrupt and the

maskable interrupts is that the maskables must be enabled by turning on the interrupt flip-flop (IFF1). The nonmaskable does not need to see IFF1 in a SET condition, and, in fact, will cause IFF1 to RESET in order to lock out the maskable interrupts ($\overline{\text{INT}}$).

The nonmaskable interrupt is very much like a "hardware restart" instruction. In fact, it is an RST 66 instruction (meaning that it will cause a restart instruction to be executed to location ØØ 66 hex). The nonmaskable interrupt cannot be disabled by software, and is always recognized by bringing. $\overline{\text{NMI}}$ LOW. Recall that the restart instructions caused program control to be transferred to one of eight locations in page zero. The principal difference between the eight software restart instructions and the nonmaskable interrupt are (1) $\overline{\text{NMI}}$ to a fixed location (address ØØ 66 hex), and (2) $\overline{\text{NMI}}$ is *hardware* implemented.

$\overline{\text{NMI}}$ is used in those situations where it is not prudent to ignore the interrupt. It may be that critical, but transitory, data may be ready to input. Or it may be an alarm condition. A program used to control the environment in a building, for example, probably would want to see no priority higher than the automatic fire alarm. One common application of $\overline{\text{NMI}}$ when the Z80 is used in a microcomputer is to guard against the problems consequent to a loss of AC mains power. A circuit is built that monitors the AC mains at the primary of the computer's DC power supply. If the AC power drops out for even a few cycles, the circuit generates an $\overline{\text{NMI}}$ signal to the CPU. The CPU will immediately honor the request, and transfer program control to a power loss subroutine. This program is used to transfer all of the data in the volatile (i.e., solid-state) memory, and the CPU registers/flags, into some form of nonvolatile memory (i.e., disc, magnetic tape, etc.). Computers that require this ability must have sufficient back-up power stored in batteries, of even the massive filter capacitors of the DC power supply, to execute the power loss subroutine before the energy gives out.

Figure 16-1 shows an example of a typical program sequence for the nonmaskable interrupt. We are executing a program in page-6 (i.e., locations from 6Ø ØØ hex). The interrupt service subroutine is stored in locations beginning at 8Ø ØØ hex. An interrupt occurs while the instruction at location 6Ø Ø3 is being executed. The sequence of events is given below:

1. $\overline{\text{NMI}}$ occurs while the CPU is executing the instruction located at 6Ø Ø3.
2. Program counter (PC) is incremented from 6Ø Ø3 to 6Ø Ø4, and then its contents are pushed onto the external memory stack.
3. PC is then loaded with ØØ 66 hex, transferring program control to ØØ 66 hex.

   Before we can service the interrupt, however, we must tend to some housekeeping chores that will allow us to reenter the main program

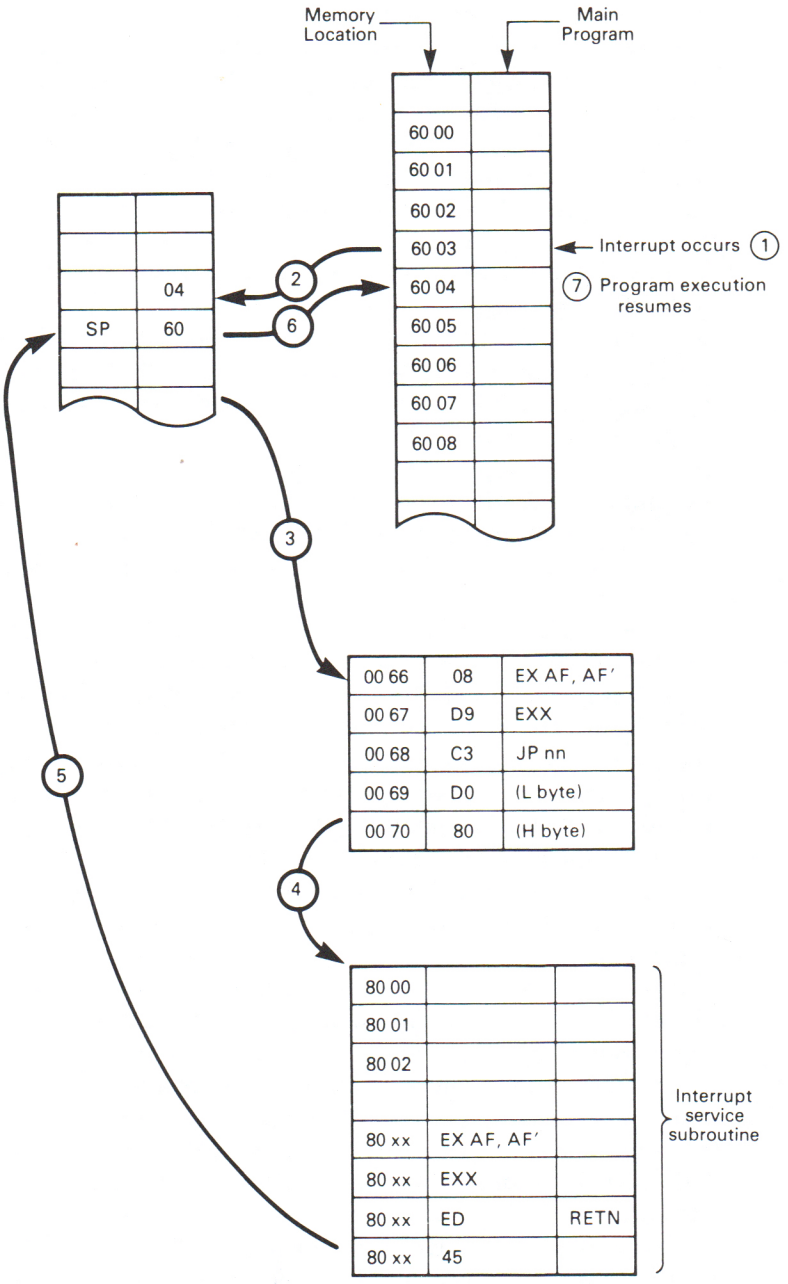


FIGURE 16-1. Operations in interrupt servicing.

at the point left off, and with no problems. We will want the main program to begin executing at the location that would have been called if the interrupt had not occurred (i.e., 6Ø Ø4). The *address* of this next location was saved automatically in an external memory stack, but nothing has been done for the flags and other CPU registers. In order to save this environment for use when program control is returned to the main program, we must execute the two exchange instructions (EX and EXX). These are the instructions located at the restart location (ØØ 66 and ØØ 67 hex). The EX instruction exchanges the contents of the AF and A′F′ registers, while the EXX instruction causes the other CPU registers to exchange with their alternates. (A′, F′, B′, C′, D′, E′, H′, and L′ are the alternate bank of CPU registers in the Z80.) The environment, i.e., the contents of the main registers, is now saved in the alternate registers. This will free the main registers for use in the interrupt subroutine, and will permit the main program to come back unscratched from the interrupt. Trying to figure out where the CPU was otherwise, without EX and EXX, is a lot like trying to nail jello to the wall.

In some cases, the interrupt service program is short enough that it can be located in the page-Ø locations following ØØ 66 hex. We could, for example, make the first instruction of the service routine at ØØ 68 hex. But we usually want to save that part of memory for other housekeeping chores (i.e., other restart instructions). In the example shown in Fig. 16-1, we execute EX and EXX to save the environment, and then jump immediately to location 8Ø ØØ.

4. The interrupt service program is located higher in memory. In this example we have located it at 8Ø ØØ. This program is not shown in detail, because its nature would depend on the type of interrupt being serviced.

The last instruction in any nonmaskable interrupt service program *must* be RETN (return from nonmaskable interrupt). This instruction tells the CPU to return control to the main program.

5. RETN returns the contents of the external memory stack to the program counter. Since the PC now contains 6Ø Ø4 hex, the program resumes at that location. This is the location immediately following the location that was executing when the $\overline{\text{NMI}}$ signal occurred.

Note that, prior to the RETN instruction, we had to re-exchange the registers by executing once again the EX and EXX instructions. This will regain the environment lost when the restart-66 occurred.

The nonmaskable interrupt is a hardware function of the Z80 CPU chip. It *cannot* be overridden by the programmer. The maskable interrupt, on the other hand, is designed so that it *can* be overridden by the programmer.

The Z80 contains two interrupt flip-flops, labelled IFF1 and IFF2. The first of these, IFF1, is the main interrupt flip-flops, whereas IFF2 is a secondary interrupt flip-flop used to store the condition of IFF1 when a nonmaskable interrupt occurs. We want the CPU restored to its previous state when the nonmaskable interrupt has been serviced. The contents of IFF1 are copied into IFF2 automatically when $\overline{\text{NMI}}$ is recognized. When RETN is executed, the contents of IFF2 are copied back to IFF1, restoring the condition of IFF1 to that existing when the interrupt occurred. This action completes the restoration of the CPU.

The $\overline{\text{NMI}}$ will automatically cause the state of IFF1 to be stored in IFF2, and then cause IFF1 to be RESET. This is done to prohibit any additional maskable interrupts during the period that $\overline{\text{NMI}}$ is being serviced.

MASKABLE INTERRUPTS

Maskable interrupts can be software-controlled through the use of EI, DI, IMØ, IM1, and IM2 instructions. The maskable interrupt is initiated by bringing the $\overline{\text{INT}}$ terminal on the Z80 (pin 16) LOW momentarily. This action is necessary, but not sufficient, to turn on the interrupt. Recall that IFF1 must be SET before a maskable interrupt is recognized by the CPU. If IFF1 is RESET, then the $\overline{\text{INT}}$ command is masked; i.e., it is not seen by the CPU—it is ignored. IFF1 is SET by executing IMØ, IM1, IM2, or EI instructions. It can RESET by applying a $\overline{\text{RESET}}$ pulse to pin 16 of the Z80, or by executing a DI (disable interrupt) instruction. There are, then, two ways to turn off the maskable interrupt capability of the CPU.

There are three types of maskable interrupts, designated mode-Ø, mode-1, and mode-2.

Mode-Ø is the "default" mode. Unless the programmer demands another mode, by causing the IM1 or IM2 instructions to be executed, mode-Ø will be assumed. The CPU is placed in mode-Ø as soon as a $\overline{\text{RESET}}$ signal is received at pin 26 of the Z80. It is usually the practice of designers to automatically apply a *power-on* $\overline{\text{RESET}}$ as soon as DC power is applied to the Z80.

Of course, setting any given interrupt mode does not allow the CPU to respond to interrupts. An EI (enable interrupt) instruction must be executed first. Once EI is executed, the interrupt flip-flop (IFF1) is SET, so the CPU will respond to $\overline{\text{INT}}$ requests (regardless of mode selected).

MODE-Ø

Mode-Ø is used to make the Z80 think that it is an 8080A microprocessor. This was probably done because one of the objectives of Z80 design was to maintain as much software compatibility between Z80 and the older 8080A as

possible. Although there are some differences where timing becomes important, it is a general rule of thumb that 8080A programs will execute on Z80 systems, but the reverse is not true; many Z80 instructions have no 8080A counterparts.

Mode-Ø is automatically selected as soon as a $\overline{\text{RESET}}$ pulse is received. Mode-Ø can also be selected through software. The IMØ instruction will cause the CPU to enter mode-Ø, and is used when the programmer has previously selected one of the other interrupt modes, and then wants to return to mode-Ø without resetting the CPU.

Like all of the maskable interrupts, mode-Ø cannot be recognized by the CPU unless the interrupt flip-flop is SET. This flip-flop will be set only if the enable interrupt (EI) instruction is executed. When this is done, then the CPU will be ready to respond to maskable interrupt requests.

The mode-Ø interrupt requires that the interrupting device place a valid Z80 instruction onto the eight-bit data bus as soon as the interrupt acknowledge signal is generated (see Chapter 11). In most cases, the instruction used is the one-byte restart instruction. There are eight unique restart instructions in the Z80 instruction repertoire, and these cause immediate jumps in program control to eight different locations in page zero.

The interrupt service routine should be located at the location in memory where the restart transfers control. For example, if a keyboard causes an interrupt and then jams a restart-10 instruction onto the data bus, the CPU will transfer control to the instruction ØØ 1Ø. If the interrupt service routine is short enough, then it might be located in the memory spots immediately following ØØ 1Ø (as well might be the case in a simple keyboard input subroutine), or the instruction may be a jump immediate to some location higher in memory. It is very common for programmers to locate these service programs in the top end of the memory available in that particular computer.

Figure 16-2 shows a typical mode-Ø response. For the sake of continuity, we are using the same locations as in the nonmaskable interrupt discussion earlier. The program is executing the instruction at location 6Ø Ø3 when the $\overline{\text{INT}}$ signal is received by the CPU. The interrupt request is recognized following the completion of the instruction at 6Ø Ø3, provided that IFF1 is SET. The sequence is as follows:

1. $\overline{\text{INT}}$ occurs during the execution of the instruction at location 6Ø Ø3. This is recognized by the CPU during the last clock cycle of that instruction.
2. On the next clock pulse, the CPU acknowledges the interrupt request by causing $\overline{\text{IORQ}}$ and $\overline{\text{M1}}$ to go LOW immediately.
3. When the interrupt acknowledge signal is received, the interrupting device places an RST 1Ø code on the CPU data bus.

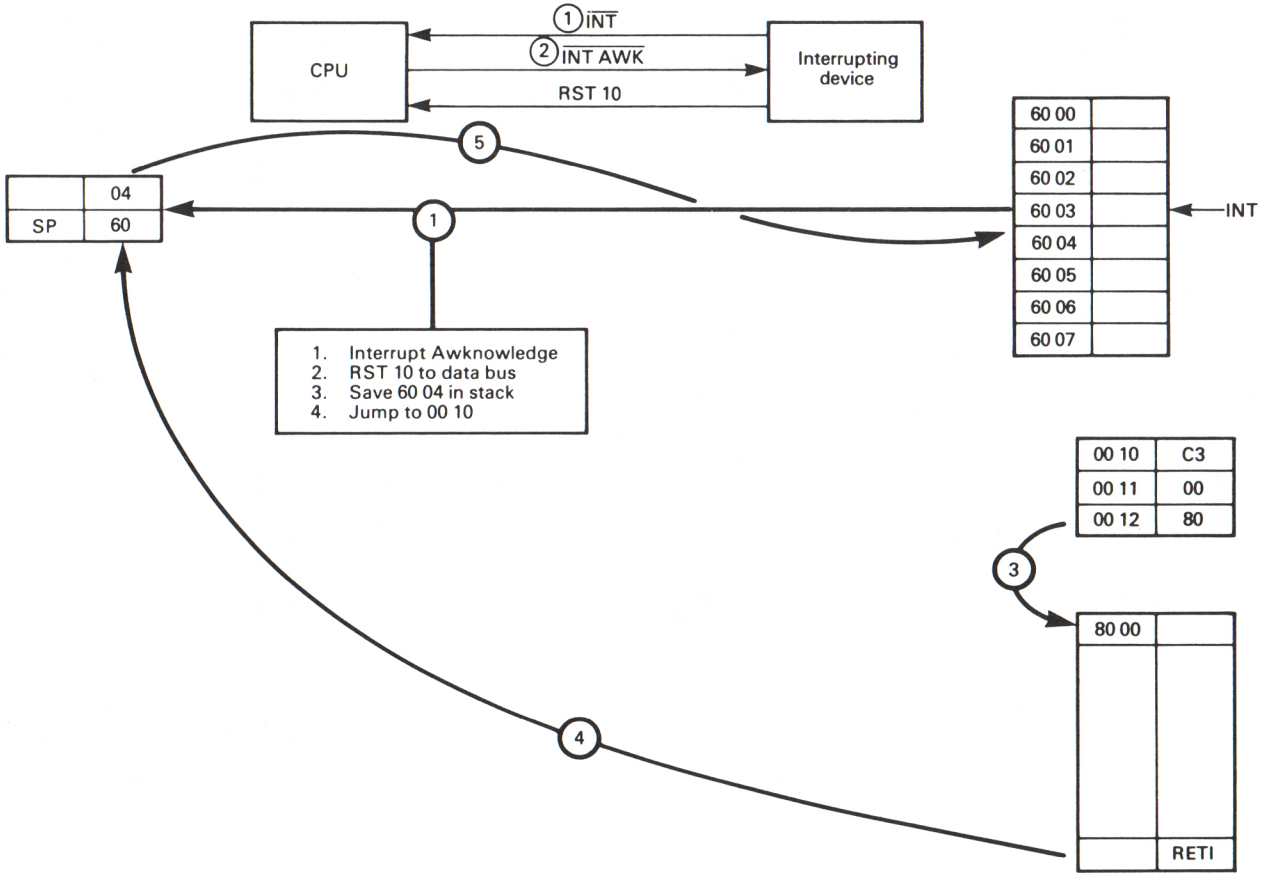

CPU
1 INT
2 INT AWK
RST 10
Interrupting device
60 00
60 01
60 02
60 03
60 04
60 05
60 06
60 07
INT
04
SP
60
1. Interrupt Awknowledge
2. RST 10 to data bus
3. Save 60 04 in stack
4. Jump to 00 10
00 10
C3
00 11
00
00 12
80
80 00
RETI
1
3
4
5

FIGURE 16-2.

4. The CPU executes the RST 1Ø by incrementing the PC to 6Ø Ø4, storing the incremented contents in the external memory stack, and then jumping immediately to location ØØ 1Ø.
5. At location ØØ 1Ø the instruction is an immediate jump to location 8Ø ØØ, where the interrupt service program is found.
6. Again, the environment must be saved. There are two ways in which this can be done. One is to use the EX and EXX instructions of the previous example. Another is to use the PUSH instructions:

```
8Ø ØØ   PUSH AF
8Ø Ø1   PUSH BC
8Ø Ø2   PUSH DE
8Ø Ø3   PUSH HL
8Ø Ø4   PUSH IY
8Ø Ø5   PUSH IX
```

The next instructions would then be instructions of the interrupt service subroutine. When this program is completed, we must execute all of the POP instructions, to bring the contents of the registers back from the memory stack, and an RETI (return from maskable interrupt) instruction:

```
8Ø xx   POP AF
8Ø xx   POP BC
8Ø xx   POP DE
8Ø xx   POP HL
8Ø xx   POP IY
8Ø xx   POP IX
8Ø xx   RETI
```

7. After the RETN instruction, the CPU will replace the contents of PC with the data stored in the external stack (6Ø Ø4). This is the address of the instruction in the main program that would have been executed *next* if the interrupt had not occurred.
8. Program execution resumes at location 6Ø Ø4.

The mode-Ø interrupt preserves some of the compatibility of the Z80 with the Intel 8080A microprocessor. But there is a limitation in this mode. The device will allow only eight interrupt devices, one for each of the eight restart locations.

Interrupt priority encoding is possible by using a priority controller, such as the Intel 8214 (or one of the related devices), or one of the Zilog Z80 peripheral chips.

MODE-1

Mode-1 is not similar to any function of the 8080A device, so it is unique to the Z80, in this respect. It is almost identical to the nonmaskable interrupt, except: (1) it is maskable, and (2) it causes a restart jump to location ØØ 38 instead of ØØ 66.

The mode-1 interrupt is dependent upon the programmer's setting mode-1 by enabling interrupt flip-flop IFF1 (the EI instruction), and setting mode-1 by executing an IM1 instruction.

The use of mode-1 is similar to the nonmaskable interrupt, except that the priority would be lower than that of a nonmaskable interrupt. It has the advantage that no external logic is needed to cause the restart instruction. It is, then, somewhat faster than the mode-Ø operation.

Refer back to the discussion of the nonmaskable interrupt for a discussion of how this interrupt is serviced. Just be sure to replace in your mind the location ØØ 66 with ØØ 38.

MODE-2

The mode-2 interrupt is one of the most powerful microcomputer interrupts. It allows vectored interrupts of up to 128 levels, as opposed to only eight levels in mode-Ø and one level in mode-1 and the nonmaskable interrupt.

Zilog has conveniently caused the Z80 peripheral control chips (Z80-PIO, Z80-SIO, and Z80-CTC) to allow prioritizing of the interrupts through a daisy-chaining scheme.

The key to the versatility of the mode-2 interrupt is that it is *vectored.* That is, it can use a single eight-bit word to point to any location in memory! The 16-bit address of the interrupt service program is stored in a table of interrupt addresses located somewhere in memory. The location of this table is pointed to by a two-byte digital word formed from the contents of the Interrupt (I) register, and the one-byte word supplied by the interrupting device. The upper eight bits of this 16-bit pointer is supplied by the I register, and must be preloaded by the program. The lower-order eight bits of the pointer is supplied by the interrupting device.

There is one restriction on the addresses of the table, and that is that they must begin on an even-numbered memory location. All of the entries in this table will be two bytes in adjacent locations. The first byte of each entry in the table is the low-order byte of the desired address, while the second entry is the high-order byte. One consequence of this constraint is that the least significant bit of the eight-bit word supplied by the interrupting device must be Ø.

Figure 16-3 shows an example of such a table. In this case, the

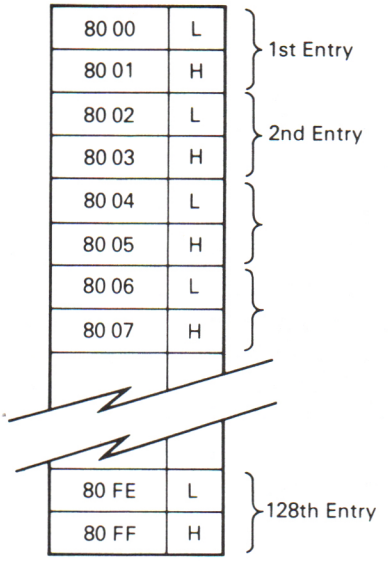


FIGURE 16-3.

programmer elected to locate the table in page-8, and it commences at 8Ø ØØ hex. The first entry is found at 8Ø ØØ and 8Ø Ø1. These locations contain the low, and high, order bytes of the address where the first interrupt service program is located. The first part of this address (8Ø hex) is stored in the I register. The second part is supplied by the interrupting device. Notice that the binary equivalent of ØØ ends in a Ø.

Similarly, the other entries are found beginning at 8Ø Ø2, 8Ø Ø4, 8Ø Ø6, etc. all the way up to 8Ø FE (if 128 levels are required). Each of these table addresses contains the address of a location in memory where the CPU will find the program that serves that particular interrupting device.

Figure 16-4 shows a typical mode-2 interrupt sequence. In this program, the main program is located in page-4 (i.e., beginning at 4Ø ØØ), the vector table is located in page-8, and the interrupt subroutine for the device shown is in page-6 (begins at 6Ø 5Ø hex). The I register contains 8Ø hex, and the interrupting device is programmed to enter Ø4 hex on the data bus when the interrupt acknowledge signal is received. The interrupt flip-flop IFF1 must be SET, and the bus request $\overline{\text{BUSRQ}}$ must be HIGH. The sequence of events is as follows:

1. The interrupting peripheral issues an $\overline{\text{INT}}$ signal to the CPU.
2. When the interrupt acknowledge signal is received, the peripheral jams Ø4 hex onto the data bus. This is merged with the 8Ø from the I register to form the address 8Ø Ø4 hex. This address in memory will

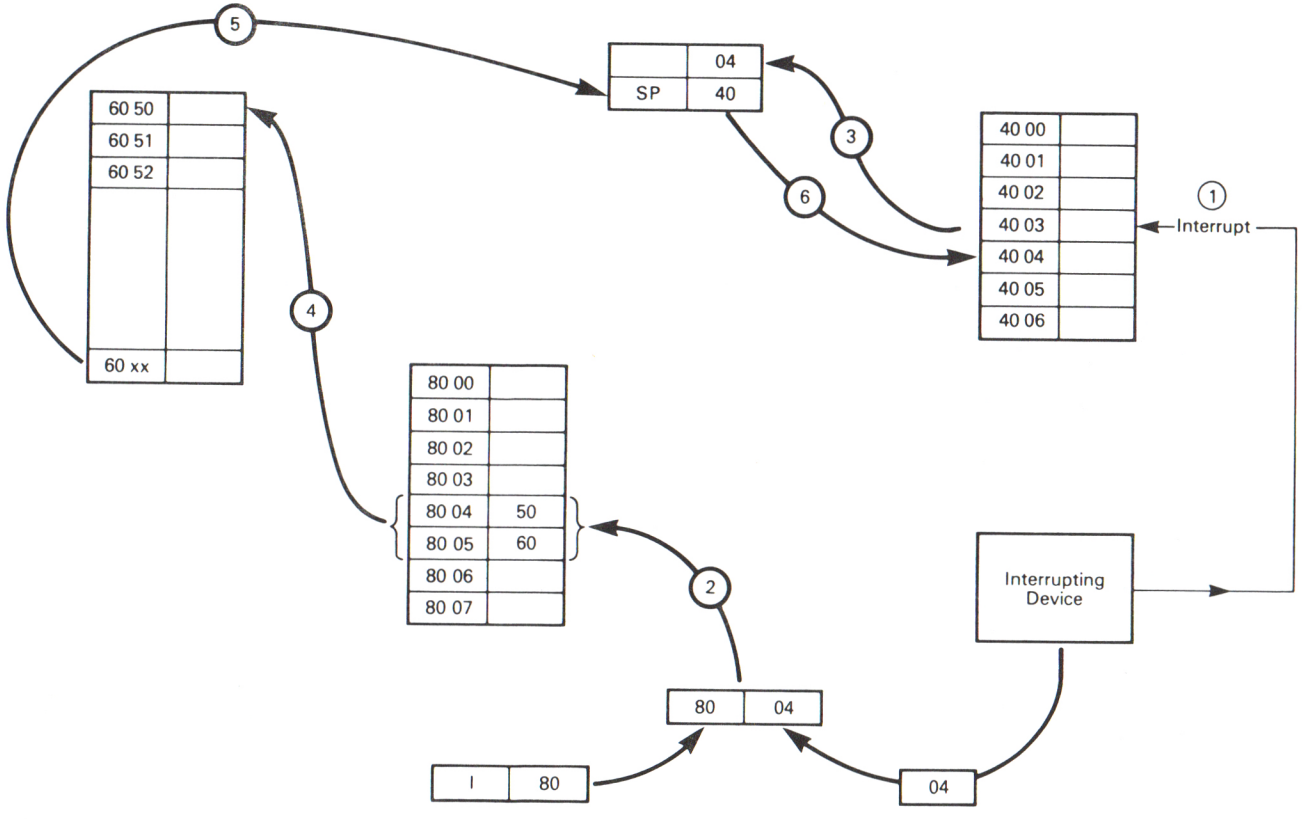

5
04
SP
40
3
6
40 00
40 01
40 02
40 03
40 04
40 05
40 06
1
Interrupt
60 50
60 51
60 52
60 xx
4
80 00
80 01
80 02
80 03
80 04
50
80 05
60
80 06
80 07
2
Interrupting Device
80
04
I
80
04

FIGURE 16-4.

contain the address of the actual interrupt service program required by this peripheral.

3. The PC is incremented, and then its contents are pushed onto an external memory stack.
4. The PC is loaded weith the address found at location 8Ø Ø4. This address is 6Ø 5Ø hex, so program control jumps to this location.
5. After the last instruction (RETI) of the service program, the PC data saved in the external stack are loaded back into the PC.
6. The main program resumes at location 4Ø Ø4.

It is necessary to save the environment when the jump occurs, or the CPU will not necessarily be in the same state as before the interrupt occurred. These techniques were discussed earlier in this chapter.

CHAPTER 17
ARITHMETIC/LOGIC OPERATIONS

The arithmetic instructions permit direct addition and subtraction operations by the Z80 CPU. By using one of the standard algorithms we are also able to provide multiplication and division operations. Note, however, that many computer users prefer to use *hardware* multiplication/division in order to gain much in the speed department. This is true of all microcomputers and most minicomputers, not just the Z80 device. The Z80 instruction set also includes a *decimal adjust* (DAA) that permits BCD arithmetic operations.

The Z80 arithmetic instructions are divided into eight- and sixteen-bit groups. Additionally, some of the control instructions also pertain to arithemtic operations.

The eight-bit group contains the following type of instructions:

ADD (addition)
ADC (addition-with-carry)
SUB (subtraction)
SBC (subtraction with carry)

Also parts of this eight-bit group are the logical instructions AND, OR, XOR, CP (compare), INC (increment), and DEC (decrement). Although these are covered elsewhere in this book, they do form a very necessary part of some arithmetic operations.

In all of the eight-bit arithmetic instructions it is assumed that the accumulator and one other operand are used. The second operand may be *immediate* (i.e., ADD A,N), the contents of one of the Z80 registers (directly or indirectly addressed), or the contents of external memory location.

The sixteen-bit arithmetic group uses register pairs (i.e., HL, IX, BC, DE, SP) in order to perform operations with longer data words. The sixteen-bit group contains ADD, ADC, and SBC instructions. It also contains INC (increment) and DEC (decrement) instructions. There are, however, no logical instructions in this group.

ADD INSTRUCTIONS

The ADD instructions permit addition between the accumulator and data obtained from another location. The result of the addition is stored in the accumulator, and certain condition flags are set to denote the nature of the result.

ADD A,A. This is a single-byte instruction that adds the contents of the accumulator to the contents of the accumulator (i.e., the contents of the accumulator are added to themselves). The result is stored in the accumulator. The op-code is 87 (hex).

Example

The contents of the accumulator are 3F (hex). If an ADD A,A instruction is encountered:

$$3F + 3F = 7E$$

The result (7E) is stored in accumulator after the execution of this instruction.

ADD A,r. This series of instructions is also single-byte. There is a separate op-code for each register of the Set B, C, D, E, H, and L. The format for these instructions op-codes is

$$1\ 0\ 0\ 0\ 0\ _\ \underline{r}\ _$$

The most significant four-bits form the hex digit "8." The second hex digit required for the op-code is determined by the particular "r" selected from the following table.

REGISTER	r
B	000
C	001
D	010
E	011
H	100
L	101

The op-codes for ADD A,r are, therefore,

B	80
C	81
D	82
E	83
H	84
L	85

When one of these instructions is encountered, then the contents of the specified register are added to the contents of the accumulator, and the result is stored in the accumulator.

ADD A, (HL). This is a register-indirect method of locating a given byte in the external memory. In this instruction, the contents of the external memory location whose address is given by the contents of the HL register pair are added to the contents of the accumulator. The result is stored in the accumulator.

Example

Consider Fig. 17-1. The contents of the accumulator are 3D (hex) when the ADD A, (HL) instruction is encountered. At the time the contents of the HL register pair are 8Ø Ø3 (pointing to location 8Ø Ø3 in external memory). The contents of the accumulator (3D) are added to the contents of 8Ø Ø3 (2B), and the results 3D + 2B, or 68 (hex), is stored in the accumulator. At the end of the execution of the ADD A, (HL) instruction the contents of the accumulator are 68.

ADD A,n. This is a two-byte instruction where the first byte is the op-code (C6) and the second byte of the value "n" to be added to the contents of the accumulator. This is an example of an immediate instruction.

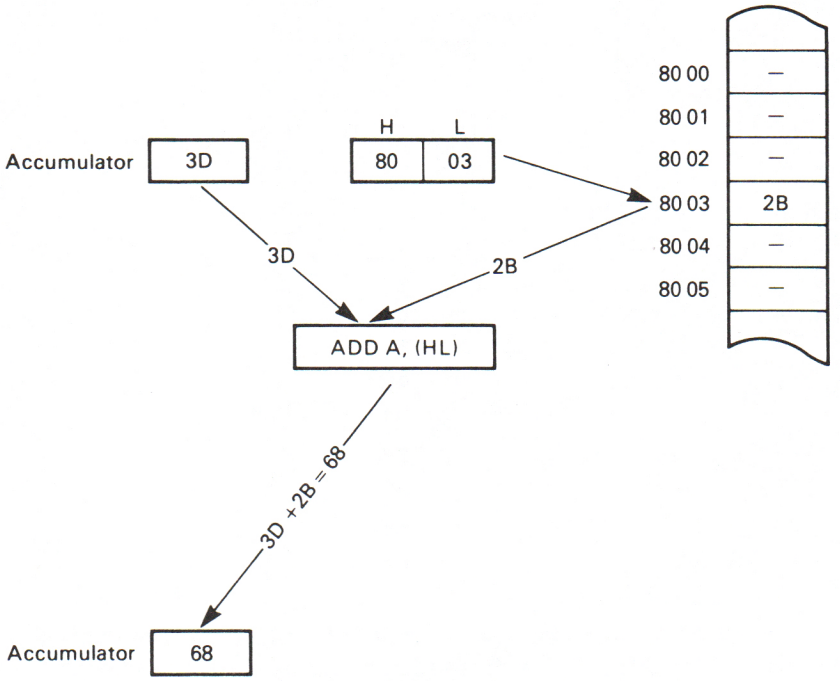


FIGURE 17-1. Operations for ADD A, (HL).

Example

The contents of the accumulator are IF when the following code is encountered:

```
C6   (op-code for ADD A,n)
3Ø   ("n")
```

This tells the Z80 to add the contents of the accumulator (1F) to "n" (3Ø). The operation carried out will be 1F + 3Ø = 4F. The value stored in the accumulator at the end of this instruction is 4F.

ADD A,(IX + *d*) and ADD A,(IY) + d). These addition instructions add together the contents of the accumulator and the contents of an external memory location that is specified by the contents of either the IY or IX index registers and a displacement integer *d*. Both of these are three-byte instructions:

IX + *d*	IY + *d*
DD	FD
86	86
d	d

ADC INSTRUCTIONS

These instructions are similar to the ADD instructions, with the exception that they take into account any carry bits generated from previous operations. The C-flag of the F register determines whether there has been a carry on any given operation.

SUB INSTRUCTIONS

The SUB instructions are analogous to the ADD instructions, and the SBC instructions are analogous to the ADC instructions. The difference, of course, is that the operation performed is subtraction instead of addition. The same range of instruction types is available as in the ADD/ADC group. Only SBC is available, however, in the sixteen-bit arithmetic group. The operations are as follows:

SBC	SUB
A ← A-s-cy	A ← A-s

LOGIC OPERATIONS

The Z80 instruction set contains certain logical and compare instructions. A logical instruction is one that performs the binary logic operations of AND, OR, or Exclusive-OR (XOR). With some exceptions, the compare instructions are functionally much like the subtraction operations.

Binary logic operations are used in digital electronics, and are similar to logic taught as philosophy in college (if P then Q). The AND function compares two inputs (i.e., bits), and issues a HIGH output (i.e., a logical-1) only if both inputs are also at a logical-1 (HIGH) level. The rules for AND are

Ø AND Ø = Ø
Ø AND 1 = Ø
1 AND Ø = Ø
1 AND 1 = 1

The OR function compares two inputs, or bits in the case of a microcomputer instruction, and issues a HIGH (logical-1) output if *either* input bit is HIGH. The rules are as follows:

Ø OR Ø = Ø
Ø OR 1 = 1
1 OR Ø = 1
1 OR 1 = 1

The Exclusive-OR, or XOR, logical function is a little different. This instruction will produce a HIGH output if either input is HIGH, but not if *both* inputs are HIGH. The rules for XOR are

Ø XOR Ø = Ø
Ø XOR 1 = 1
1 XOR Ø = 1
1 XOR 1 = Ø

Note that both cases where the output is HIGH (perhaps we should say the *result*) are when the two bits (inputs) are the *same*. In the case of Ø XOR Ø and 1 XOR 1, the output is LOW (logical-Ø).

The Z80 microprocessor provides all three types of logical operation (AND, OR, XOR). All of these are actually multi-instruction functions AND s, OR s, and XOR s.

The AND s instructions include AND R, AND N, AND (HL), AND (IX + d), and AND (IY + d). Like the other forms of Z80 logical instruction, AND s performs a *bit-by-bit* comparison between the specified data and the contents of the accumulator. The result is stored in the accumulator.

The AND R instruction performs an AND operation between the accumulator and the register specified by a three-bit field in the op-code. The op-code is of the form 10100---, where the last three bits specify the register from the set ØØØ (B), ØØ1 (C), Ø1Ø (D), Ø11 (E), 1ØØ (H), 1Ø1 (L), and 111 (A, or accumulator).

The AND N instruction is an immediate type in which the AND operation is performed between the accumulator and the byte following the op-code. The op-code is 11100110 (E6 hex), while the word to be compared with the accumulator is the following byte. For example, suppose the accumulator contains the byte 5F hex, and the following code is encountered:

E6 op-code for AND N
D7 data (hex for 215_{10})

This means that we want to perform an AND operation between 5F hex and D7 hex. On a bit-for-bit basis this is

```
      Ø 1 Ø 1 1 1 1 1     (5F hex)
AND   1 1 0 1 0 1 1 1     (D7 hex)
      ---------------
      0 1 0 1 0 1 1 1     (57 hex)
```

(see the rules given earlier, and apply them on a bit-by-bit basis without any carries or borrows).

This means that 5F hex AND D7 hex, when compared in an AND instruction, produce a result of 57 hex, and this result is left in the accumulator at the end of the instruction's execution.

The AND instruction is used to mask out any unwanted bits in the eight-bit data word stored in the accumulator. We might, for example, make use of bit-packing to make more efficient use of the field. We might, for example, want to see only bits 3, 4, and 5 (i.e., bits 3-5 out of the field Ø-7). This means that we will need to mask out bits Ø, 1, 2, 6, and 7. By placing a 1 at all bits that we want to keep, and a Ø at all other bit positions, the desired bits will be passed. Recall the rules for AND. We will compare the desired bits with a 1. If the input bit is 1, then we have a 1 AND 1 = 1 situation. But if we input a Ø, then we have a Ø AND 1 = Ø situation. This means that the sense of the input bit is retained, while all others are masked out.

For example, in the situation given above, we wanted to mask out all bits except bits 3-5, so we would want to compare the contents input to the accumulator with ØØ111ØØØ. All bits except 3, 4, and 5 are zero, so will be masked out regardless of the input word. Suppose that the contents of the accumulator are 1Ø1Ø1Ø1Ø, and we want to retain only bits 3-5. We would use the AND N instruction as follows:

```
1 1 1 Ø Ø 1 1 Ø     (E6 hex)

Ø Ø 1 1 1 Ø Ø Ø     (N)
```

The AND comparison performed by this, when the contents of the accumulator are 10101010, will be:

```
      Ø Ø 1 1 1 Ø Ø Ø
AND   1 Ø 1 Ø 1 Ø 1 Ø
      ---------------
      Ø Ø 1 Ø 1 Ø Ø Ø
```

Note that in the Z80 some of the bit-test instructions can also be used to strip the field of the desired bits.

The other AND instructions use memory locations pointed to by the contents of certain registers. AND (HL), for example, compares the contents of the accumulator with the contents of an external memory location pointed to by the contents of the HL register pair. The AND (IY + d) and AND (IX + d) instructions perform the comparison between the accumulator and external memory locations pointed to by a displacement integer *d*, which is part of the instruction, and either the IY or IX index registers.

The OR instruction is also of the OR s format, similar to the AND s instructions discussed above. The same different types of OR instruction are allowed: OR R, OR N, OR (HL), OR (YI + d) and OR (IX + d). We use the OR s instructions to merge data, or to set certain bits.

Recall that the OR rules require a 1 result if *either* "input" is 1. For example, let us compare B3 hex with C9 hex in an OR instructions:

```
     1 0 1 1 0 0 1 1     (B3 hex)
OR   1 1 0 0 1 0 0 1     (C9 hex)
     ---------------
     1 1 1 1 1 0 1 1     (FB hex)
```

Let us consider an example of the case where we want to set bit-7 to 1 unconditionally. We would OR the contents of the accumulator with 1ØØØØØØØ. This would make bit 7 a 1 regardless of the data:

```
     1 Ø Ø Ø Ø Ø Ø Ø
OR   1 Ø 1 1 Ø Ø Ø 1
     ---------------
     1 Ø 1 1 Ø Ø Ø 1
Bit 7 is 1 ——↗

     1 Ø Ø Ø Ø Ø Ø Ø
OR   Ø 1 1 Ø 1 Ø Ø 1
     ---------------
     1 1 1 Ø 1 Ø Ø 1
Bit 7 is 1 ——↗
```

The XOR s instruction performs a bit-by-bit Exclusive-OR operation between the contents of the accumulator and the contents of the source, *s*. We have XOR R, XOR N, XOR (HL), XOR (IY +d), and XOR (IX +d) instructions in the Z80 set.

Recall the rules for the XOR operation. We will create a 1 result only if both inputs are different from each other (i.e., Ø XOR 1, or 1 XOR Ø). An example of an XOR operation using the XOR N instruction follows.

Suppose the accumulator contains B2 hex, and we encounter the following XOR N instruction:

```
1 1 1 Ø 1 1 1 Ø   (EE hex—op-code for XOR N)
Ø Ø 1 Ø 1 Ø 1 Ø   (2A hex—N)
```

The operation that the Z80 will perform is

```
        Ø Ø 1 Ø 1 Ø 1 Ø     (2A hex)
XOR     1 Ø 1 1 Ø Ø 1 Ø     (B2 hex)
      ---------------------
        1 Ø Ø 1 1 Ø Ø Ø     (98 hex)
```

When we compare 2A hex with B2 hex in an XOR instruction, the result is 98 hex, and this is stored in the accumulator at the end of the execution of the XOR N instruction.

CHAPTER 18

MACHINE LANGUAGE PROGRAMMING OF THE T/S-1000

The Timex/Sinclair microcomputers are programmed mainly in the BASIC language. In fact, the bulk of the Timex instruction manual is given over to BASIC programming, and only a very small mention in Chapter 25 deals with machine language programming at all. This is somewhat lamentable, because machine language can be a lot more efficient than high-level languages like BASIC. Of course, it takes a lot of effort to make a machine language program work properly, but the result is a program that takes less space, and is able to execute a lot faster than BASIC programs.

There are a couple of ways to use machine language programs. In one scenario, we might embed the machine language program into some larger BASIC program through the use of the USR function. Alternatively, we might begin executing the machine language program immediately and more or less forget the BASIC language. Of course, the nature of the Timex operating system is such that we cannot totally ignore the BASIC interpreter.

There are several constraints on the Timex machine language programmer that may not exist on other Z80-based machines. We have to work around that ROM-based operating system, so will find that this machine doesn't behave exactly like a *tabula rasa* Z80 machine. According to the T/S-1000 instruction book we have the following constraints on the use of the USR subroutine:

1. The value 1E (hex) must be in the *I* register on return from the USR function.
2. Avoid using the *A'*, *F'*, *IX*, *IY* and *R* registers in your machine language program, as these are used in the display program.

ENTERING MACHINE CODE

Machine code is entered into the T/S-1000 using the PEEK and POKE instructions. The POKE instruction allows you to insert a piece of data at a specific memory location. For example, suppose you want to insert the number "78" into location 17,301. The code would be written:

```
POKE 17301,78
```

This instruction will place the number 78 into memory location 17301. If you want to examine that location to determine the data at that point, then type

```
PRINT PEEK 17301
```

and the computer will display the number at that location, in this case "78." The numbers used in PEEK and POKE must be between −255 and +255.

USING THE REM STATEMENT

The BASIC REM statement is used normally to write documentary text on the screen during LIST operations, and is not normally executed. When machine language programming the T/S-1000 we can use dummy REM statements to keep the BASIC system from overwriting our machine language program. Character strings are also used for this purpose. When using the REM statement, type in a REM line with sufficient characters (even dummy characters like periods, etc.) to hold the machine code of the machine language subroutine. These can then be POKEd into memory.

MACHINE LANGUAGE ON Z80 MICROCOMPUTERS

The most successful machine-language programmers are those who learn the instruction in detail, as well as the addressing modes of the computer (see Chapter 14) and the way the machine handles operations like arithmetic and logic (AND, OR, XOR and so forth).

In this book, we have given detailed information concerning those matters that affect all Z8-based computers, including the T/S-1000. With a little practice, you should be able to use this information to write rather extensive machine language programming either for the T/S-1000, or, for more advanced computers when (if?) you decide to "trade-up." Note that in almost all cases machine language programs will transfer from one machine to another provided that both use the same microprocessor. Thus, you will be able to run the programs you write for this computer on other computers. The only problem will be formatting and memory allocations, which may have to be changed on other machines.

I personally believe that the best solution will be the use of an Assembler such as that offered by BUG-BYTE, called ZXAS. An assembler program will allow you to program in Z80 mnemonics, rather than difficult to handle machine language hexadecimal numbers.

I also recommend that you obtain *Mastering Machine Code on Your ZX-81* by Toni Baker (Reston, VA.: Reston Publishing Company, 1983). This book was written for the ZX-81 but is equally valid for the T/S-1000.

CHAPTER 19

DETAILS OF THE Z80 INSTRUCTION SET

SUMMARY OF FLAG OPERATION

Instruction	D7 S	Z		H		P/V	N	D0 C	Comments
ADD A, s; ADC A, s	↕	↕	X	↕	X	V	0	↕	8-bit add or add with carry
SUB s; SBC A, s; CP s; NEG	↕	↕	X	↕	X	V	1	↕	8-bit subtract, subtract with carry, compare and negate accumulator
AND s	↕	↕	X	1	X	P	0	0	Logical operations
OR s; XOR s	↕	↕	X	0	X	P	0	0	
INC s	↕	↕	X	↕	X	V	0	•	8-bit increment
DEC s	↕	↕	X	↕	X	V	1	•	8-bit decrement
ADD DD, SS	•	•	X	X	X	•	0	↕	16-bit add
ADC HL, SS	↕	↕	X	X	X	V	0	↕	16-bit add with carry
SBC HL, SS	↕	↕	X	X	X	V	1	↕	16-bit subtract with carry
RLA; RLCA; RRA; RRCA	•	•	X	0	X	•	0	↕	Rotate accumulator
RL s; RLC s; RR s; RRC s; SLA s; SRA s; SRL s	↕	↕	X	0	X	P	0	↕	Rotate and shift locations
RLD; RRD	↕	↕	X	0	X	P	0	•	Rotate digit left and right
DAA	↕	↕	X	↕	X	P	•	↕	Decimal adjust accumulator
CPL	•	•	X	1	X	•	1	•	Complement accumulator
SCF	•	•	X	0	X	•	0	1	Set carry
CCF	•	•	X	X	X	•	0	↕	Complement carry
IN r, (C)	↕	↕	X	0	X	P	0	•	Input register indirect
INI; IND; OUTI; OUTD	X	↕	X	X	X	X	1	•	Block input and output
INIR; INDR; OTIR; OTDR	X	1	X	X	X	X	1	•	Z = 0 if B ≠ 0 otherwise Z = 1
LDI; LDD	X	X	X	0	X	↕	0	•	Block transfer instructions
LDIR; LDDR	X	X	X	0	X	0	0	•	P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI; CPIR; CPD; CPDR	X	↕	X	X	X	↕	1	•	Block search instructions Z = 1 if A = (HL), otherwise Z = 0 P/V = 1 if BC ≠ 0, otherwise P/V = 0
LD A, I; LD A, R	↕	↕	X	0	X	IFF	0	•	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s	X	↕	X	1	X	X	0	•	The state of bit b of location s is copied into the Z flag

The following notation is used in this table:

Symbol	Operation
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract. H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
↕	The flag is affected according to the result of the operation.
•	The flag is unchanged by the operation.
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care".
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8-bit location for all the addressing modes allowed for the particular instruction.
ss	Any 16-bit location for all the addressing modes allowed for that instruction.
ii	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range <0, 255>
nn	16-bit value in range <0, 65535>

8-BIT LOAD GROUP
'LD'

DESTINATION		SOURCE: IMPLIED I	IMPLIED R	REGISTER A	B	C	D	E	H	L	REG INDIRECT (HL)	(BC)	(DE)	INDEXED (IX+d)	(IY+d)	EXT. ADDR. (nn)	IMME. n
REGISTER	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
REGISTER	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
REGISTER	C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
REGISTER	D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
REGISTER	E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
REGISTER	H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
REGISTER	L			6F	68	69	6A	6B	6C	6D	6E			DD 6E d	FD 6E d		2E n
REG INDIRECT	(HL)			77	70	71	72	73	74	75							36 n
REG INDIRECT	(BC)			02													
REG INDIRECT	(DE)			12													
INDEXED	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
INDEXED	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
EXT. ADDR.	(nn)			32 n n													
IMPLIED	I			ED 47													
IMPLIED	R			ED 4F													

8-BIT LOAD GROUP

Mnemonic	Symbolic Operation	Flags S	Z	H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments r, s	Reg.
LD r, s	r ← s	•	•	X	•	X	•	•	•	01 r s		1	1	4	r, s	Reg.
LD r, n	r ← n	•	•	X	•	X	•	•	•	00 r 110		2	2	7	000	B
										← n →					001	C
LD r, (HL)	r ← (HL)	•	•	X	•	X	•	•	•	01 r 110		1	2	7	010	D
LD r, (IX+d)	r ← (IX+d)	•	•	X	•	X	•	•	•	11 011 101	DD	3	5	19	011	E
										01 r 110					100	H
										← d →					101	L
LD r, (IY+d)	r ← (IY+d)	•	•	X	•	X	•	•	•	11 111 101	FD	3	5	19	111	A
										01 r 110						
										← d →						
LD (HL), r	(HL) ← r	•	•	X	•	X	•	•	•	01 110 r		1	2	7		
LD (IX+d), r	(IX+d) ← r	•	•	X	•	X	•	•	•	11 011 101	DD	3	5	19		
										01 110 r						
										← d →						
LD (IY+d), r	(IY+d) ← r	•	•	X	•	X	•	•	•	11 111 101	FD	3	5	19		
										01 110 r						
										← d →						
LD (HL), n	(HL) ← n	•	•	X	•	X	•	•	•	00 110 110	36	2	3	10		
										← n →						
LD (IX+d), n	(IX+d) ← n	•	•	X	•	X	•	•	•	11 011 101	DD	4	5	19		
										00 110 110	36					
										← d →						
										← n →						
LD (IY+d), n	(IY+d) ← n	•	•	X	•	X	•	•	•	11 111 101	FD	4	5	19		
										00 110 110	36					
										← d →						
										← n →						
LD A, (BC)	A ← (BC)	•	•	X	•	X	•	•	•	00 001 010	0A	1	2	7		
LD A, (DE)	A ← (DE)	•	•	X	•	X	•	•	•	00 011 010	1A	1	2	7		
LD A, (nn)	A ← (nn)	•	•	X	•	X	•	•	•	00 111 010	3A	3	4	13		
										← n →						
										← n →						
LD (BC), A	(BC) ← A	•	•	X	•	X	•	•	•	00 000 010	02	1	2	7		
LD (DE), A	(DE) ← A	•	•	X	•	X	•	•	•	00 010 010	12	1	2	7		
LD (nn), A	(nn) ← A	•	•	X	•	X	•	•	•	00 110 010	32	3	4	13		
										← n →						
										← n →						
LD A, I	A ← I	↕	↕	X	0	X	IFF	0	•	11 101 101	ED	2	2	9		
										01 010 111	57					
LD A, R	A ← R	↕	↕	X	0	X	IFF	0	•	11 101 101	ED	2	2	9		
										01 011 111	5F					
LD I, A	I ← A	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	9		
										01 000 111	47					
LD R, A	R ← A	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	9		
										01 001 111	4F					

Notes: r, s means any of the registers A, B, C, D, E, H, L
IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

16-BIT LOAD GROUP
'LD'
'PUSH' AND 'POP'

SOURCE

DESTINATION		REGISTER AF	BC	DE	HL	SP	IX	IY	IMM. EXT. nn	EXT. ADDR. (nn)	REG. INDIR. (SP)
REGISTER	AF										F1
	BC								01 n n	ED 4B n n	C1
	DE								11 n n	ED 5B n n	D1
	HL								21 n n	2A n n	E1
	SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
	IX								DD 21 n n	DD 2A n n	DD E1
	IY								FD 21 n n	FD 2A n n	FD E1
EXT. ADDR.	(nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
REG. IND.	(SP)	F5	C5	D5	E5		DD E5	FD E5			

PUSH INSTRUCTIONS →

NOTE: The Push & Pop Instructions adjust the SP after every execution.

↑ POP INSTRUCTIONS

16-BIT LOAD GROUP

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
LD dd, nn	dd ← nn	•	•	X	•	X	•	•	•	00 dd0 001 ← n → ← n →		3	3	10	dd Pair 00 BC 01 DE
LD IX, nn	IX ← nn	•	•	X	•	X	•	•	•	11 011 101 00 100 001 ← n → ← n →	DD 21	4	4	14	10 HL 11 SP
LD IY, nn	IY ← nn	•	•	X	•	X	•	•	•	11 111 101 00 100 001 ← n → ← n →	FD 21	4	4	14	
LD HL, (nn)	H ← (nn+1) L ← (nn)	•	•	X	•	X	•	•	•	00 101 010 ← n → ← n →	2A	3	5	16	
LD dd, (nn)	dd_H ← (nn+1) dd_L ← (nn)	•	•	X	•	X	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	ED	4	6	20	
LD IX, (nn)	IX_H ← (nn+1) IX_L ← (nn)	•	•	X	•	X	•	•	•	11 011 101 00 101 010 ← n → ← n →	DD 2A	4	6	20	
LD IY, (nn)	IY_H ← (nn+1) IY_L ← (nn)	•	•	X	•	X	•	•	•	11 111 101 00 101 010 ← n → ← n →	FD 2A	4	6	20	
LD (nn), HL	(nn+1) ← H (nn) ← L	•	•	X	•	X	•	•	•	00 100 010 ← n → ← n →	22	3	5	16	
LD (nn), dd	(nn+1) ← dd_H (nn) ← dd_L	•	•	X	•	X	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	ED	4	6	20	
LD (nn), IX	(nn+1) ← IX_H (nn) ← IX_L	•	•	X	•	X	•	•	•	11 011 101 00 100 010 ← n → ← n →	DD 22	4	6	20	
LD (nn), IY	(nn+1) ← IY_H (nn) ← IY_L	•	•	X	•	X	•	•	•	11 111 101 00 100 010 ← n → ← n →	FD 22	4	6	20	
LD SP, HL	SP ← HL	•	•	X	•	X	•	•	•	11 111 001	F9	1	1	6	
LD SP, IX	SP ← IX	•	•	X	•	X	•	•	•	11 011 101 11 111 001	DD F9	2	2	10	
LD SP, IY	SP ← IY	•	•	X	•	X	•	•	•	11 111 101 11 111 001	FD F9	2	2	10	
PUSH qq	(SP-2) ← qq_L (SP-1) ← qq_H	•	•	X	•	X	•	•	•	11 qq0 101		1	3	11	qq Pair 00 BC 01 DE
PUSH IX	(SP-2) ← IX_L (SP-1) ← IX_H	•	•	X	•	X	•	•	•	11 011 101 11 100 101	DD E5	2	4	15	10 HL 11 AF
PUSH IY	(SP-2) ← IY_L (SP-1) ← IY_H	•	•	X	•	X	•	•	•	11 111 101 11 100 101	FD E5	2	4	15	
POP qq	qq_H ← (SP+1) qq_L ← (SP)	•	•	X	•	X	•	•	•	11 qq0 001		1	3	10	
POP IX	IX_H ← (SP+1) IX_L ← (SP)	•	•	X	•	X	•	•	•	11 011 101 11 100 001	DD E1	2	4	14	
POP IY	IY_H ← (SP+1) IY_L ← (SP)	•	•	X	•	X	•	•	•	11 111 101 11 100 001	FD E1	2	4	14	

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
e.g. BC_L = C, AF_H = A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ flag is affected according to the result of the operation.

EXCHANGES 'EX' AND 'EXX'

		IMPLIED ADDRESSING				
		AF'	BC', DE' & HL'	HL	IX	IY
IMPLIED	AF	08				
	BC, DE & HL		no			
	DE			EB		
REG. INDIR.	(SP)			E3	DD E3	FD E3

BLOCK TRANSFER GROUP

		SOURCE	
		REG. INDIR. (HL)	
DESTINATION	REG. INDIR. (DE)	ED A0	'LDI' – Load (DE) ← (HL) Inc HL & DE, Dec BC
		ED B0	'LDIR' – Load (DE)←(HL) Inc HL & DE, Dec BC, Repeat until BC = 0
		ED A8	'LDD' – Load (DE) ← (HL) Dec HL & DE, Dec BC
		ED B8	'LDDR' – Load (DE)←(HL) Dec HL & DE, Dec BC, Repeat until BC = 0

HL points to source
DE points to destination
BC is byte counter

BLOCK SEARCH GROUP

SEARCH LOCATION	
REG. INDIR. (HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR' – Inc HL, Dec BC repeat until BC = 0 or find match
ED A9	'CPD' – Dec HL & BC
ED B9	'CPDR' – Dec HL & BC Repeat until BC = 0 or find match

HL points to location in memory to be compared with accumulator contents
BC is byte counter

EXCHANGE GROUP AND BLOCK TRANSFER AND SEARCH GROUP

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE↔HL	•	•	X	•	X	•	•	•	11 101 011	EB	1	1	4	
EX AF, AF'	AF↔AF'	•	•	X	•	X	•	•	•	00 001 000	08	1	1	4	
EXX	BC↔BC', DE↔DE', HL↔HL'	•	•	X	•	X	•	•	•	11 011 001	D9	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H↔(SP+1) L↔(SP)	•	•	X	•	X	•	•	•	11 100 011	E3	1	5	19	
EX (SP), IX	IX_H↔(SP+1) IX_L↔(SP)	•	•	X	•	X	•	•	•	11 011 101 11 100 011	DD E3	2	6	23	
EX (SP), IY	IY_H↔(SP+1) IY_L↔(SP)	•	•	X	•	X	•	•	•	11 111 101 11 100 011	FD E3	2	6	23	
LDI	(DE)←(HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	X	0	X	①↕	0	•	11 101 101 10 100 000	ED A0	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE)←(HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 Repeat until BC = 0	•	•	X	0	X	0	0	•	11 101 101 10 110 000	ED B0	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE)←(HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	X	0	X	①↕	0	•	11 101 101 10 101 000	ED A8	2	4	16	
LDDR	(DE)←(HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 Repeat until BC = 0	•	•	X	0	X	0	0	•	11 101 101 10 111 000	ED B8	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A – (HL) HL ← HL+1 BC ← BC-1	↕	②↕	X	↕	X	①↕	1	•	11 101 101 10 100 001	ED A1	2	4	16	
CPIR	A – (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	↕	②↕	X	↕	X	①↕	1	•	11 101 101 10 110 001	ED B1	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A – (HL) HL ← HL-1 BC ← BC-1	↕	②↕	X	↕	X	①↕	1	•	11 101 101 10 101 001	ED A9	2	4	16	
CPDR	A – (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	↕	②↕	X	↕	X	①↕	1	•	11 101 101 10 111 001	ED B9	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

8-BIT ARITHMETIC AND LOGIC

SOURCE

	REGISTER ADDRESSING							REG. INDIR.	INDEXED		IMMED.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
'ADD'	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
ADD w CARRY 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SUBTRACT 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SUB w CARRY 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
'AND'	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
COMPARE 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENT 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENT 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

8-BIT ARITHMETIC AND LOGICAL GROUP

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD A, r	A ← A + r	↕	↕	X	↕	X	V	0	↕	10 [000] r		1	1	4	r Reg.
ADD A, n	A ← A + n	↕	↕	X	↕	X	V	0	↕	11 [000] 110 ← n →		2	2	7	000 B 001 C 010 D
ADD A, (HL)	A ← A+(HL)	↕	↕	X	↕	X	V	0	↕	10 [000] 110		1	2	7	011 E
ADD A, (IX+d)	A ← A+(IX+d)	↕	↕	X	↕	X	V	0	↕	11 011 101 10 [000] 110 ← d →	DD	3	5	19	100 H 101 L 111 A
ADD A, (IY+d)	A ← A+(IY+d)	↕	↕	X	↕	X	V	0	↕	11 111 101 10 [000] 110 ← d →	FD	3	5	19	
ADC A, s	A ← A+s+CY	↕	↕	X	↕	X	V	0	↕	[001]					s is any of r, n, (HL), (IX+d), (IY+d) as shown for ADD instruction. The indicated bits replace the [000] in the ADD set above.
SUB s	A ← A - s	↕	↕	X	↕	X	V	1	↕	[010]					
SBC A, s	A ← A - s - CY	↕	↕	X	↕	X	V	1	↕	[011]					
AND s	A ← A ∧ s	↕	↕	X	1	X	P	0	0	[100]					
OR s	A ← A ∨ s	↕	↕	X	0	X	P	0	0	[110]					
XOR s	A ← A ⊕ s	↕	↕	X	0	X	P	0	0	[101]					
CP s	A - s	↕	↕	X	↕	X	V	1	↕	[111]					
INC r	r ← r + 1	↕	↕	X	↕	X	V	0	•	00 r [100]		1	1	4	
INC (HL)	(HL) ← (HL)+1	↕	↕	X	↕	X	V	0	•	00 110 [100]		1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	↕	↕	X	↕	X	V	0	•	11 011 101 00 110 [100] ← d →	DD	3	6	23	
INC (IY+d)	(IY+d) ← (IY+d)+1	↕	↕	X	↕	X	V	0	•	11 111 101 00 110 [100] ← d →	FD	3	6	23	
DEC s	s ← s - 1	↕	↕	X	↕	X	V	1	•	[101]					s is any of r, (HL), (IX+d), (IY+d) as shown for INC. DEC same format and states as INC. Replace [100] with [101] in OP Code.

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow, P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

GENERAL PURPOSE AF OPERATIONS

Decimal Adjust Acc, 'DAA'	27
Complement Acc, 'CPL'	2F
Negate Acc, 'NEG' (2's complement)	ED 44
Complement Carry Flag, 'CCF'	3F
Set Carry Flag, 'SCF'	37

MISCELLANEOUS CPU CONTROL

'NOP'	00	
'HALT'	76	
DISABLE INT '(DI)'	F3	
ENABLE INT '(EI)'	FB	
SET INT MODE 0 'IM 0'	ED 46	8080A MODE
SET INT MODE 1 'IM 1'	ED 56	RESTART TO LOCATION 0038_H
SET INT MODE 2 'IM 2'	ED 5E	INDIRECT CALL USING REGISTER I AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER.

GENERAL PURPOSE ARITHMETIC AND CPU CONTROL GROUPS

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
DAA	Converts acc, content into packed BCD following add or subtract with packed BCD operands	‡	‡	X	‡		X	P	•	‡	00 100 111	27	1	1	4	Decimal adjust accumulator
CPL	$A - \overline{A}$	•	•	X	1		X	•	1	•	00 101 111	2F	1	1	4	Complement accumulator (One's complement)
NEG	$A - \overline{A} + 1$	‡	‡	X	‡		X	V	1	‡	11 101 101 01 000 100	ED 44	2	2	8	Negate acc, (two's complement)
CCF	$CY - \overline{CY}$	•	•	X	X		X	•	0	‡	00 111 111	3F	1	1	4	Complement carry flag
SCF	CY - 1	•	•	X	0		X	•	0	1	00 110 111	37	1	1	4	Set carry flag
NOP	No operation	•	•	X	•		X	•	•	•	00 000 000	00	1	1	4	
HALT	CPU halted	•	•	X	•		X	•	•	•	01 110 110	76	1	1	4	
DI *	IFF - 0	•	•	X	•		X	•	•	•	11 110 011	F3	1	1	4	
EI *	IFF - 1	•	•	X	•		X	•	•	•	11 111 011	FB	1	1	4	
IM 0	Set interrupt mode 0	•	•	X	•		X	•	•	•	11 101 101 01 000 110	ED 46	2	2	8	
IM 1	Set interrupt mode 1	•	•	X	•		X	•	•	•	11 101 101 01 010 110	ED 56	2	2	8	
IM 2	Set interrupt mode 2	•	•	X	•		X	•	•	•	11 101 101 01 011 110	ED 5E	2	2	8	

Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.
* = Interrupts are not sampled at the end of EI or DI

16-BIT ARITHMETIC

DESTINATION		SOURCE	BC	DE	HL	SP	IX	IY
	'ADD'	HL	09	19	29	39		
		IX	DD 09	DD 19		DD 39	DD 29	
		IY	FD 09	FD 19		FD 39		FD 29
	ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
	SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
	INCREMENT 'INC'		03	13	23	33	DD 23	FD 23
	DECREMENT 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

16-BIT ARITHMETIC GROUP

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD HL, ss	HL ← HL+ss	•	•	X	X	X	•	0	‡	00 ss1 001		1	3	11	ss Reg. 00 BC
ADC HL, ss	HL ← HL+ss+CY	‡	‡	X	X	X	V	0	‡	11 101 101 01 ss1 010	ED	2	4	15	01 DE 10 HL 11 SP
SBC HL, ss	HL ← HL-ss-CY	‡	‡	X	X	X	V	1	‡	11 101 101 01 ss0 010	ED	2	4	15	
ADD IX, pp	IX ← IX + pp	•	•	X	X	X	•	0	‡	11 011 101 00 pp1 001	DD	2	4	15	pp Reg. 00 BC 01 DE 10 IX 11 SP
ADD IY, rr	IY ← IY + rr	•	•	X	X	X	•	0	‡	11 111 101 00 rr1 001	FD	2	4	15	rr Reg. 00 BC 01 DE 10 IY 11 SP
INC ss	ss ← ss + 1	•	•	X	•	X	•	•	•	00 ss0 011		1	1	6	
INC IX	IX ← IX + 1	•	•	X	•	X	•	•	•	11 011 101 00 100 011	DD 23	2	2	10	
INC IY	IY ← IY + 1	•	•	X	•	X	•	•	•	11 111 101 00 100 011	FD 23	2	2	10	
DEC ss	ss ← ss - 1	•	•	X	•	X	•	•	•	00 ss1 011		1	1	6	
DEC IX	IX ← IX - 1	•	•	X	•	X	•	•	•	11 011 101 00 101 011	DD 2B	2	2	10	
DEC IY	IY ← IY - 1	•	•	X	•	X	•	•	•	11 111 101 00 101 011	FD 2B	2	2	10	

Notes: ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

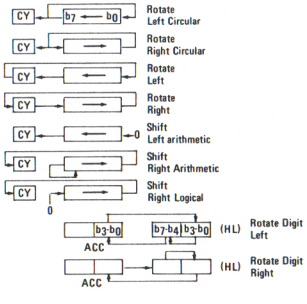
Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
‡ = flag is affected according to the result of the operation.

ROTATES AND SHIFTS

Source and Destination

TYPE OF ROTATE OR SHIFT	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
'RLC'	CB 07	CB 00	CB 01	CB 02	CB 03	CB 04	CB 05	CB 06	DD CB d 06	FD CB d 06
'RRC'	CB 0F	CB 08	CB 09	CB 0A	CB 0B	CB 0C	CB 0D	CB 0E	DD CB d 0E	FD CB d 0E
'RL'	CB 17	CB 10	CB 11	CB 12	CB 13	CB 14	CB 15	CB 16	DD CB d 16	FD CB d 16
'RR'	CB 1F	CB 18	CB 19	CB 1A	CB 1B	CB 1C	CB 1D	CB 1E	DD CB d 1E	FD CB d 1E
'SLA'	CB 27	CB 20	CB 21	CB 22	CB 23	CB 24	CB 25	CB 26	DD CB d 26	FD CB d 26
'SRA'	CB 2F	CB 28	CB 29	CB 2A	CB 2B	CB 2C	CB 2D	CB 2E	DD CB d 2E	FD CB d 2E
'SRL'	CB 3F	CB 38	CB 39	CB 3A	CB 3B	CB 3C	CB 3D	CB 3E	DD CB d 3E	FD CB d 3E
'RLD'								ED 6F		
'RRD'								ED 67		

	A
'RLCA'	07
'RRCA'	0F
'RLA'	17
'RRA'	1F



ROTATE AND SHIFT GROUP

Mnemonic	Symbolic Operation	S	Z	H	P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
RLCA	CY ← 7 ← 0 ← A	•	•	X	0	X	•	0	‡	00 000 111	07	1	1	4	Rotate left circular accumulator
RLA	CY ← 7 ← 0 ← A	•	•	X	0	X	•	0	‡	00 010 111	17	1	1	4	Rotate left accumulator
RRCA	7 → 0 → CY, A	•	•	X	0	X	•	0	‡	00 001 111	0F	1	1	4	Rotate right circular accumulator
RRA	7 → 0 → CY, A	•	•	X	0	X	•	0	‡	00 011 111	1F	1	1	4	Rotate right accumulator
RLC r		‡	‡	X	0	X	P	0	‡	11 001 011 00 000 r	CB	2	2	8	Rotate left circular register r r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC (HL)		‡	‡	X	0	X	P	0	‡	11 001 011 00 000 110	CB	2	4	15	
RLC (IX+d)	CY ← 7 ← 0 ← r,(HL),(IX+d),(IY+d)	‡	‡	X	0	X	P	0	‡	11 011 101 11 001 011 ← d → 00 000 110	DD CB	4	6	23	
RLC (IY+d)		‡	‡	X	0	X	P	0	‡	11 111 101 11 001 011 ← d → 00 000 110	FD CB	4	6	23	
RL s	CY ← 7 ← 0 ← s≡r,(HL),(IX+d),(IY+d)	‡	‡	X	0	X	P	0	‡	010					Instruction format and states are as shown for RLC's. To form new Op-Code replace 000 of RLC's with shown code
RRC s	7 → 0 → CY, s≡r,(HL),(IX+d),(IY+d)	‡	‡	X	0	X	P	0	‡	001					
RR s	7 → 0 → CY, s≡r,(HL),(IX+d),(IY+d)	‡	‡	X	0	X	P	0	‡	011					
SLA s	CY ← 7 ← 0 ← 0, s≡r,(HL),(IX+d),(IY+d)	‡	‡	X	0	X	P	0	‡	100					
SRA s	7 → 0 → CY, s≡r,(HL),(IX+d),(IY+d)	‡	‡	X	0	X	P	0	‡	101					
SRL s	0 → 7 → 0 → CY, s≡r,(HL),(IX+d),(IY+d)	‡	‡	X	0	X	P	0	‡	111					
RLD	A 7-4 3-0 ↔ 7-4 3-0 (HL)	‡	‡	X	0	X	P	0	•	11 101 101 01 101 111	ED 6F	2	5	18	Rotate digit left and right between the accumulator and location (HL). The content of the upper half of the accumulator is unaffected
RRD	A 7-4 3-0 ↔ 7-4 3-0 (HL)	‡	‡	X	0	X	P	0	•	11 101 101 01 100 111	ED 67	2	5	18	

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ‡ = flag is affected according to the result of the operation.

BIT MANIPULATION GROUP

	BIT	REGISTER ADDRESSING							REG. INDIR.	INDEXED	
		A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
TEST 'BIT'	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	CB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
RESET BIT 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
SET BIT 'SET'	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

BIT SET, RESET AND TEST GROUP

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States
BIT b, r	$Z - \overline{r_b}$	X	‡	X	1	X	X	0	•	11 001 011 01 b r	CB	2	2	8
BIT b, (HL)	$Z - \overline{(HL)_b}$	X	‡	X	1	X	X	0	•	11 001 011 01 b 110	CB	2	3	12
BIT b, (IX+d)	$Z - \overline{(IX+d)_b}$	X	‡	X	1	X	X	0	•	11 011 101 11 001 011 - d - 01 b 110	DD CB	4	5	20
BIT b, (IY+d)	$Z - \overline{(IY+d)_b}$	X	‡	X	1	X	X	0	•	11 111 101 11 001 011 - d - 01 b 110	FD CB	4	5	20
SET b, r	$r_b - 1$	•	•	X	•	X	•	•	•	11 001 011 $\boxed{11}$ b r	CB	2	2	8
SET b, (HL)	$(HL)_b - 1$	•	•	X	•	X	•	•	•	11 001 011 $\boxed{11}$ b 110	CB	2	4	15
SET b, (IX+d)	$(IX+d)_b - 1$	•	•	X	•	X	•	•	•	11 011 101 11 001 011 - d - $\boxed{11}$ b 110	DD CB	4	6	23
SET b, (IY+d)	$(IY+d)_b - 1$	•	•	X	•	X	•	•	•	11 111 101 11 001 011 - d - $\boxed{11}$ b 110	FD CB	4	6	23
RES b, s	$s_b - 0$ $s \equiv r, (HL), (IX+d), (IY+d)$	•	•	X	•	X	•	•	•	$\boxed{10}$				

Comments	
r	Reg.
000	B
001	C
010	D
011	E
100	H
101	L
111	A
b	Bit Tested
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

To form new O_P Code replace $\boxed{11}$ of SET b, s with $\boxed{10}$. Flags and time states for SET instruction

Notes: The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ‡ = flag is affected according to the result of the operation.

JUMP GROUP

CONDITION

			UN-COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG.	SIGN POS.	REG. B≠0
JUMP 'JP'	IMMED. EXT.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVE	PC + e	18 e 2	38 e 2	30 e 2	28 e 2	20 e 2					
JUMP 'JP'	REG. INDIR.	(HL)	E9									
JUMP 'JP'		(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
DECREMENT B, JUMP IF NON ZERO 'DJNZ'	RELATIVE	PC + e										10 e 2

JUMP GROUP

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No.of M Cycles	No.of T States	Comments
JP nn	PC ← nn	•	•	X	•	X	•	•	•	11 000 011 ← n ← ← n ←	C3	3	3	10	
JP cc, nn	If condition cc is true PC ← nn, otherwise continue	•	•	X	•	X	•	•	•	11 cc 010 ← n ← ← n ←		3	3	10	cc — Condition 000 NZ non zero 001 Z zero 010 NC non carry 011 C carry 100 PO parity odd 101 PE parity even 110 P sign positive 111 M sign negative
JR e	PC ← PC + e	•	•	X	•	X	•	•	•	00 011 000 ← e-2 ←	18	2	3	12	
JR C, e	If C = 0, continue	•	•	X	•	X	•	•	•	00 111 000 ← e-2 ←	38	2	2	7	If condition not met
	If C = 1, PC ← PC+e											2	3	12	If condition is met
JR NC, e	If C = 1, continue	•	•	X	•	X	•	•	•	00 110 000 ← e-2 ←	30	2	2	7	If condition not met
	If C = 0, PC ← PC+e											2	3	12	If condition is met
JR Z, e	If Z = 0 continue	•	•	X	•	X	•	•	•	00 101 000 ← e-2 ←	28	2	2	7	If condition not met
	If Z = 1, PC ← PC+e											2	3	12	If condition is met
JR NZ, e	If Z = 1, continue	•	•	X	•	X	•	•	•	00 100 000 ← e-2 ←	20	2	2	7	If condition not met
	If Z = 0, PC ← PC+e											2	3	12	If condition is met
JP (HL)	PC ← HL	•	•	X	•	X	•	•	•	11 101 001	E9	1	1	4	
JP (IX)	PC ← IX	•	•	X	•	X	•	•	•	11 011 101 11 101 001	DD E9	2	2	8	
JP (IY)	PC ← IY	•	•	X	•	X	•	•	•	11 111 101 11 101 001	FD E9	2	2	8	
DJNZ, e	B ← B-1 If B = 0, continue	•	•	X	•	X	•	•	•	00 010 000 ← e-2 ←	10	2	2	8	If B = 0
	If B ≠ 0, PC ← PC+e											2	3	13	If B ≠ 0

Notes: e represents the extension in the relative addressing mode.
e is a signed two's complement number in the range <-126, 129>
e-2 in the op-code provides an effective address of pc+e as PC is incremented by 2 prior to the addition of e.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

CALL AND RETURN GROUP

			CONDITION									
			UN-COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG.	SIGN POS.	REG. B ≠ 0
'CALL'	IMMED. EXT.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
RETURN 'RET'	REGISTER INDIR.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REGISTER INDIR.	(SP) (SP+1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REGISTER INDIR.	(SP) (SP+1)	ED 45									

NOTE – CERTAIN FLAGS HAVE MORE THAN ONE PURPOSE. REFER TO Z80-CPU TECHNICAL MANUAL FOR DETAILS.

RESTART GROUP

		OP CODE	
CALL ADDRESS	0000_H	C7	'RST 0'
	0008_H	CF	'RST 8'
	0010_H	D7	'RST 16'
	0018_H	DF	'RST 24'
	0020_H	E7	'RST 32'
	0028_H	EF	'RST 40'
	0030_H	F7	'RST 48'
	0038_H	FF	'RST 56'

CALL AND RETURN GROUP

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
CALL nn	(SP-1) ← PC_H (SP-2) ← PC_L PC ← nn	•	•	X	•	X	•	•	•	11 001 101 ← n → ← n →	CD	3	5	17	
CALL cc, nn	If condition cc is false continue, otherwise same as CALL nn	•	•	X	•	X	•	•	•	11 cc 100 ← n → ← n →		3 3	3 5	10 17	If cc is false If cc is true
RET	PC_L ← (SP) PC_H ← (SP+1)	•	•	X	•	X	•	•	•	11 001 001	C9	1	3	10	
RET cc	If condition cc is false continue, otherwise same as RET	•	•	X	•	X	•	•	•	11 cc 000		1 1	1 3	5 11	If cc is false If cc is true
RETI	Return from interrupt	•	•	X	•	X	•	•	•	11 101 101 01 001 101	ED 4D	2	4	14	
RETN[1]	Return from non maskable interrupt	•	•	X	•	X	•	•	•	11 101 101 01 000 101	ED 45	2	4	14	
RST p	(SP-1) ← PC_H (SP-2) ← PC_L PC_H ← 0 PC_L ← p	•	•	X	•	X	•	•	•	11 t 111		1	3	11	

cc	Condition	
000	NZ	non zero
001	Z	zero
010	NC	non carry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

t	p
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

[1] RETN loads IFF_2 → IFF_1

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

INPUT GROUP

INPUT DESTINATION		PORT ADDRESS IMMED. n	PORT ADDRESS REG. INDIR. (C)
INPUT 'IN'	REG. ADDRESSING A	DB n	ED 78
	B		ED 40
	C		ED 48
	D		ED 50
	E		ED 58
	H		ED 60
	L		ED 68
'INI' – INPUT & Inc HL, Dec B	REG. INDIR. (HL)		ED A2
'INIR' – INP, Inc HL, Dec B, REPEAT IF B≠0			ED B2
'IND' – INPUT & Dec HL, Dec B			ED AA
'INDR'–INPUT, Dec HL Dec B, REPEAT IF B≠0			ED BA

BLOCK INPUT COMMANDS

OUTPUT GROUP

'OUT'			SOURCE A	B	C	D	E	H	L	REG. IND. (HL)
	IMMED.	n	D3 n							
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
'OUTI' – OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3
'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0	REG. IND.	(C)								ED B3
'OUTD' – OUTPUT Dec HL Dec B	REG. IND.	(C)								ED AB
'OTDR'–OUTPUT, Dec HL Dec B, REPEAT IF B ≠ 0	REG. IND.	(C)								ED BB

BLOCK OUTPUT COMMANDS

PORT DESTINATION ADDRESS

INPUT AND OUTPUT GROUP

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
IN A, (n)	A ← (n)	•	•	X	•	X	•	•	•	11 011 011 - n -	DB	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
IN r, (C)	r ← (C) if r = 110 only the flags will be affected	↕	↕	X	↕	X	P	0	•	11 101 101 01 r 000	ED	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INI	(HL) ← (C) B ← B - 1 HL ← HL + 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 100 010	ED A2	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INIR	(HL) ← (C) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 110 010	ED B2	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
IND	(HL) ← (C) B ← B - 1 HL ← HL - 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 101 010	ED AA	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INDR	(HL) ← (C) B ← B - 1 HL ← HL - 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 111 010	ED BA	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUT (n), A	(n) ← A	•	•	X	•	X	•	•	•	11 010 011 - n -	D3	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	•	•	X	•	X	•	•	•	11 101 101 01 r 001	ED	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTI	(C) ← (HL) B ← B - 1 HL ← HL + 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 100 011	ED A3	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTIR	(C) ← (HL) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 110 011	ED B3	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTD	(C) ← (HL) B ← B - 1 HL ← HL - 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 101 011	ED AB	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTDR	(C) ← (HL) B ← B - 1 HL ← HL - 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 111 011	ED BB	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$

Notes: ① If the result of B - 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ↕ = flag is affected according to the result of the operation.

Z80 – CPU INTERRUPT STRUCTURE

MASKABLE ($\overline{INT}$)

Mode 0

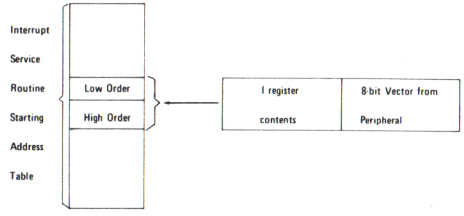
Place instruction onto Data Bus during $\overline{INTA} = \overline{MI} \bullet \overline{IORQ}$ like 8080A

Mode 1

Restart to 38_H or 56_{10} ('RST 56')

Mode 2

Used by Z80 Peripherals



NON MASKABLE ($\overline{NMI}$)

Restart to 66_H or 102_{10}

INTERRUPT ENABLE/DISABLE FLIP-FLOPS

Action	IFF_1	IFF_2	
CPU Reset	0	0	
DI	0	0	
EI	1	1	
LD A, I	•	•	IFF_2 - Parity flag
LD A, R	•	•	IFF_2 - Parity flag
Accept $\overline{NMI}$	0	•	
RETN	IFF_2	•	IFF_2 - IFF_1
Accept $\overline{INT}$	0	0	
RETI	•	•	

"•" indicates no change

PIO PROGRAMMING SUMMARY

REGISTER SELECTION

SELECT LINES C/D	SELECT LINES B/A	REGISTER SELECTED
0	0	A Data
0	1	B Data
1	0	A Control
1	1	B Control

LOAD INTERRUPT VECTOR

D7							D0	
V7	V6	V5	V4	V3	V2	V1	0	Control Register

SET OPERATING MODE

D7							D0	
M1	M0	X	X	1	1	1	1	Control Register

Mode Number	M1	M0	Mode
0	0	0	Output
1	0	1	Input
2	1	0	Bidirectional
3	1	1	Bit Control

If Mode 3 selected, the next control word to the PIO is

D7							D0	
I/O_7	I/O_6	I/O_5	I/O_4	I/O_3	I/O_2	I/O_1	I/O_0	Control Register

I/O = 1 Sets bit to Input

I/O = 0 Sets bit to Output

SET INTERRUPT CONTROL

D7							D0	
Int Enable	AND/ OR	High/ Low	Mask Follows	0	1	1	1	Control Register

In Mode 3 if Mask follows = 1, the next control word to the PIO is

D7							D0	
MB_7	MB_6	MB_5	MB_4	MB_3	MB_2	MB_1	MB_0	Control Register

MB = 0 Monitor the bit

MB = 1 Mask the bit

ENABLE / DISABLE INTERRUPTS

D7							D0	
Int Enable	X	X	X	0	0	1	1	Control Register

CTC PROGRAMMING SUMMARY

REGISTER SELECTION

SELECT LINES CS_1	CS_0	CHANNEL SELECTED	PRIORITY
0	0	0	Highest
0	1	1	
1	0	2	
1	1	3	Lowest

READ = DOWN COUNTER WRITE = CONTROL REGISTER

LOAD INTERRUPT VECTOR

$CS_0 = CS_1 = 0$

D7							D0	
V_7	V_6	V_5	V_4	V_3	X	X	0	Control Register

XX is the binary equivalent of interrupting channel number

SET OPERATING MODE

Timer Mode only

D7							D0	
Interrupt Enable	Mode	Range	Slope	Trigger	Load Time Constant	Reset	1	Control Register
	Counter/Timer	256/16	+/–	On/Off				

If Load Time Constant = 1 the next control word is the Time Constant:

D7							D0
TC_7	TC_6	TC_5	TC_4	TC_3	TC_2	TC_1	TC_0

CTC Channel interrupts when 01_H is decremented to 00_H

Time Content	Decimal counts to interrupt
01_H	1
.	.
.	.
FF_H	255
00_H	256

SIO PROGRAMMING SUMMARY

CHANNEL SELECTION

C/D	B/A	FUNCTION
0	0	Channel A Data
0	1	Channel B Data
1	0	Channel A Commands/Status
1	1	Channel B Commands/Status

READ REGISTERS

READ REGISTER 0

D7	D6	D5	D4	D3	D2	D1	D0

- D0 — Rx CHARACTER AVAILABLE
- D1 — INT PENDING
- D2 — Tx BUFFER EMPTY
- D3 — DCD
- D4 — SYNC/HUNT
- D5 — CTS
- D6 — SENDING CRC/SYNCS
- D7 — BREAK/ABORT

READ REGISTER 2*

D7	D6	D5	D4	D3	D2	D1	D0

- D0 — V0
- D1 — V1
- D2 — V2
- D3 — V3
- D4 — V4
- D5 — V5
- D6 — V6
- D7 — V7

INTERRUPT VECTOR

*Can Only Be Read By Channel B

READ REGISTER 1

D7	D6	D5	D4	D3	D2	D1	D0

- D0 — ALL SENT
- D1 — PARITY ERROR
- D2 — Rx OVERRUN ERROR
- D3 — CRC/FRAMING ERROR
- D4 — END OF FRAME (SDLC)

D4	D3	D2	I-FIELD BITS IN PREVIOUS BYTE	I-FIELD BITS IN SECOND PREVIOUS BYTE
1	0	0	0	3
0	1	0	0	4
1	1	0	0	5
0	0	1	0	6
1	0	1	0	7
0	1	1	0	8
1	1	1	1	8
0	0	0	2	8

*RESIDUE DATA

WRITE REGISTERS

WRITE REGISTER 0

D7	D6	D5	D4	D3	D2	D1	D0	
					0	0	0	REGISTER 0
					0	0	1	REGISTER 1
					0	1	0	REGISTER 2
					0	1	1	REGISTER 3
					1	0	0	REGISTER 4
					1	0	1	REGISTER 5
					1	1	0	REGISTER 6
					1	1	1	REGISTER 7
			0	0	0			NULL CODE
			0	0	1			SEND ABORT (SDLC)
			0	1	0			RESET EXT. STATUS INTERRUPTS
			0	1	1			CHANNEL RESET
			1	0	0			RESET RxINT ON FIRST CHARACTER
			1	0	1			RESET RxINT PENDING
			1	1	0			ERROR RESET
			1	1	1			RETURN FROM INT (CH-A-ONLY)
0	0							NULL CODE
0	1							RESET Rx CRC CHECKER
1	0							RESET Tx CRC GENERATOR
1	1							RESET CRC/SYNCS SENT/SENDING LATCH

WRITE REGISTER 1

D7	D6	D5	D4	D3	D2	D1	D0	
								EXT. INT ENABLE
								Tx INT ENABLE
								STATUS AFFECTS VECTOR
					0	0		Rx INT ENABLE
					0	1		Rx INT ON FIRST CHARACTER ONLY ERROR
					1	0		INT ON ALL Rx CHARACTERS (PARITY AFFECTS VECTOR)
					1	1		INT ON ALL Rx CHARACTERS (PARITY DOES NOT AFFECT VECTOR)
								WAIT/READY ON R/T
								WAIT FN/READY FN
								WAIT/READY ENABLE

WRITE REGISTER 2*

D7	D6	D5	D4	D3	D2	D1	D0	
								V0
								V1
								V2
								V3
								V4
								V5
								V6
								V7

INTERRUPT VECTOR

*Can Only Be Written By Channel B

WRITE REGISTER 3

D7	D6	D5	D4	D3	D2	D1	D0	
								Rx ENABLE
								SYNC CHARACTER LOAD INHIBIT
								ADDRESS SEARCH *MODE (SDLC)
								Rx CRC ENABLE
								ENTER HUNT MODE
								AUTO ENABLES
0	0							Rx 5 BITS/CHARACTER
0	1							Rx 7 BITS/CHARACTER
1	0							Rx 6 BITS/CHARACTER
1	1							Rx 8 BITS/CHARACTER

SIO PROGRAMMING SUMMARY (Cont'd.)

WRITE REGISTER 4

D7	D6	D5	D4	D3	D2	D1	D0	
								PARITY ENABLE
								PARITY EVEN/ODD
				0	0			SYNC MODES ENABLE
				0	1			1 STOP BIT/CHARACTER
				1	0			1½ STOP BITS/CHARACTER
				1	1			2 STOP BITS/CHARACTER
			0	0				8 BITS SYNC CHARACTER
			0	1				16 BIT SYNC CHARACTER
			1	0				SDLC MODE (01111110 SYNC FLAG)
			1	1				EXTERNAL SYNC MODE
0	0							X1 CLOCK MODE
0	1							X16 CLOCK MODE
1	0							X32 CLOCK MODE
1	1							X64 CLOCK MODE

WRITE REGISTER 5

D7	D6	D5	D4	D3	D2	D1	D0	
								Tx CRC ENABLE
								RTS
								SDLC/CRC 16
								Tx ENABLE
								SEND BREAK
				0	0			Tx 5 BITS (OR LESS) CHARACTER
				0	1			Tx 6 BITS/CHARACTER
				1	0			Tx 7 BITS/CHARACTER
				1	1			Tx 8 BITS/CHARACTER
								DTR

WRITE REGISTER 6

D7	D6	D5	D4	D3	D2	D1	D0	
								SYNC BIT 0
								SYNC BIT 1
								SYNC BIT 2
								SYNC BIT 3
								SYNC BIT 4 *
								SYNC BIT 5
								SYNC BIT 6
								SYNC BIT 7

*ALSO SDLC ADDRESS FIELD

WRITE REGISTER 7

D7	D6	D5	D4	D3	D2	D1	D0	
								SYNC BIT 8
								SYNC BIT 9
								SYNC BIT 10
								SYNC BIT 11 *
								SYNC BIT 12
								SYNC BIT 13
								SYNC BIT 14
								SYNC BIT 15

*FOR SDLC IT MUST BE PROGRAMMED TO "01111110" FOR FLAG RECOGNITION

STATUS AFFECTS VECTOR (D_2) (FROM WRITE REG 1)

If this mode is selected, the vector returned from an interrupt acknowledge cycle will be variable according to the following:

	V_3	V_2	V_1	
Ch B	0	0	0	Ch B Transmit Buffer Empty
	0	0	1	Ch B External/Status Change
	0	1	0	Ch B Receive Character Available
	0	1	1	Ch B Special Receive Condition
Ch A	1	0	0	Ch A Transmit Buffer Empty
	1	0	1	Ch A External/Status Change
	1	1	0	Ch A Receive Character Available
	1	1	1	Ch A Special Receive Condition

If this bit is 0, the fixed vector programmed in the vector register is returned.

APPENDIX A
MISCELLANEOUS Z80 INFORMATION

The instruction set (and certain other information) is rarely given in easily available form. Programmers often have to look up certain information in several different books. In this chapter, we are going to help you by offering certain information that has proved useful in programming any computer. Included is the Z80 instruction set sorted (a) by op-code, and (b) by mnemonic. The former will help you especially when you are trying to decode a program, whereas the latter is of use most often when one is trying to originate a new program.

ASCII (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE) CODE

This code is used on most modern printers and is commonly employed in many peripheral devices. It is a seven-bit code, so in eight-bit systems one additional bit is available. On keyboards, and certain other peripherals, the eighth bit is used to strobe the computer. The computer will continuously loop, polling the input, to determine the state of the eighth bit (usually designated B7). It will branch to an input instruction only if the strobe bit indicates that the data are valid. Typically, B7 is made HIGH-active. When B7 goes HIGH, therefore, the computer knows that the data are valid, so an input operation is executed. But when B7 is LOW, it is assumed that the data are invalid (i.e., garbage).

Table A-1 CPU INSTRUCTIONS SORTED BY OP-CODE

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
00	NOP	282E	JR Z, DIS	50	LD D, B
018405	LD BC, NN	29	ADD HL, HL	51	LD D, C
02	LD (BC), A	2A8405	LD (HL), (NN)	52	LD D, D
03	INC BC	2B	DEC HL	53	LD D, E
04	INC B	2C	INC L	54	LD D, H
05	DEC B	2D	DEC L	55	LD D, L
0620	LD B, N	2E20	LD L, N	56	LD D, (HL)
07	RLCA	2F	CPL	57	LD D, A
08	EX AF, AF′	302E	JR NC, DIS	58	LD E, B
09	ADD HL, BC	318405	LD SP, NN	59	LD E, C
0A	LD A,(BC)	328405	LD (NN), A	5A	LD E, D
0B	DEC BC	33	INC SP	5B	LD E, E
0C	INC C	34	INC (HL)	5C	LD E, H
0D	DEC C	35	DEC (HL)	5D	LD E, L
0E20	LD C, N	3620	LD (HL), N	5E	LD E, (HL)
0F	RRCA	37	SCF	5F	LD E, A
102E	DJNZ DIS	382E	JR C, DIS	60	LD H, B
118405	LD DE, NN	39	ADD HL, SP	61	LD H, C
12	LD (DE), A	3A8405	LD A, (NN)	62	LD H, D
13	INC DE	3B	DEC SP	63	LD H, E
14	INC D	3C	INC A	64	LD H, H
15	DEC D	3D	DEC A	65	LD H, L
1620	LD D, N	3E20	LD A, N	66	LD H, (HL)
17	RLA	3F	CCF	67	LD H, A
182E	JR DIS	40	LD B, B	68	LD L, B
19	ADD HL, DE	41	LD B, C	69	LD L, C
1A	LD A,(DE)	42	LD B, D	6A	LD L, D
1B	DEC DE	43	LD B, E	6B	LD L, E
1C	INC E	44	LD B, H, NN	6C	LD L, H
1D	DEC E	45	LD B, L	6D	LD L, L
1E20	LD E, N	46	LD B, (HL)	6E	LD L, (HL)
1F	RRA	47	LD B, A	6F	LD L, A
202E	JR NZ, DIS	48	LD C, B	70	LD (HL), B
218405	LD HL, NN	49	LD C, C	71	LD (HL), C
228405	LD (NN), HL	4A	LD C, D	72	LD (HL), D
23	INC HL	4B	LD C, E	73	LD (HL), E
24	INC H	4C	LD C, H	74	LD (HL), H
25	DEC H	4D	LD C, L	75	LD (HL), L
2620	LD H, N	4E	LD C, (HL)	76	HALT
27	DAA	4F	LD C, A	77	LD (HL), A

Table A-1 (cont'd.)

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
78	LD A, B	A0	AND B	C8	RET Z
79	LD A, C	A1	AND C	C9	RET
7A	LD A, D	A2	AND D	CA8405	JP Z, NN
7B	LD A, E	A3	AND E	CC8405	CALL Z, NN
7C	LD A, H	A4	AND H	CD8405	CALL NN
7D	LD A, L	A5	AND L	CE20	ADC A, N
7E	LD A, (HL)	A6	AND (HL)	CF	RST 8
7F	LD A, A	A7	AND A	D0	RET NC
80	ADD A, B	A8	XOR B	D1	POP DE
81	ADD A, C	A9	XOR C	D28405	JP NC, NN
82	ADD A, D	AA	XOR D	D320	OUT (N), A
83	ADD A, E	AB	XOR E	D48405	CALL NC, NN
84	ADD A, H	AC	XOR H	D5	PUSH DE
85	ADD A, L	AD	XOR L	D620	SUB N
86	ADD A, (HL)	AE	XOR (HL)	D7	RST 10H
87	ADD A, A	AF	XOR A	D8	RET C
88	ADC A, B	B0	OR B	D9	EXX
89	ADC A, C	B1	OR C	DA8405	JP C, NN
8A	ADC A, D	B2	OR D	DB20	IN A, (N)
8B	ADC A, E	B3	OR E	DC8405	CALL C, N
8C	ADC A, H	B4	OR H	DE20	SBC A, N
8D	ADC A, L	B5	OR L	DF	RST 18H
8E	ADC A, (HL)	B6	OR (HL)	E0	RET PO
8F	ADC A, A	B7	OR A	E1	POP HL
90	SUB B	B8	CP B	E28405	JP PO, NN
91	SUB C	B9	CP C	E3	EX (SP), HL
92	SUB D	BA	CP D	E48405	CALL PO, NN
93	SUB E	BB	CP E	E5	PUSH HL
94	SUB H	BC	CP H	E620	AND N
95	SUB L	BD	CP L	E7	RST 20 H
96	SUB (HL)	BE	CP (HL)	E8	RET PE
97	SUB A	BF	CP A	E9	JP (HL)
98	SBC A, B	C0	RET NZ	EA8405	JE PE NN
99	SBC A, C	C1	POP BC	EB	EX DE, HL
9A	SBC A, D	C28405	JP NZ, NN	EC8405	CALL PE, NN
9B	SBC A, E	C38405	JP NN	EE20	XOR N
9C	SBC A, H	C48405	CALL NZ, NN	EF	RST 28H
9D	SBC A, L	C5	PUSH BC	F0	RET P
9E	SBC A, (HL)	C620	ADD A, N	F1	POP AF
9F	SBC A, A	C7	RST O	F28405	JP P, NN

Table A-1 (cont'd.)

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
F3	D1	CB1C	RR H	CB4C	BIT 1, H
F48405	CALL P, NN	CB1D	RRL	CB4D	BIT 1, L
F5	PUSH AF	CB1E	RR (HL)	CB4E	BIT 1, (HL)
F620	OR N	CB1F	RR A	CB4F	BIT 1, A
F7	RST 30H	CB20	SLA B	CB50	BIT 2, B
F8	RET M	CB21	SLA C	CB51	BIT 2, C
F9	LD SP, HL	CB22	SLA D	CB52	BIT 2, D
FA8405	JP M, NN	CB23	SLA E	CB53	BIT 2, E
FB	E1	CB24	SLA H	CB54	BIT 2, H
FC8405	CALL M, NN	CB25	SLA L	CB55	BIT 2, L
FE20	CP N	CB26	SLA (HL)	CB56	BIT 2, (HL)
FF	RST 38H	CB27	SLA A	CB57	BIT 2, A
CB00	RLC B	CB28	SRA B	CB58	BIT 3, B
CB01	RLC C	CB29	SRA C	CB59	BIT 3, C
CB02	RLC D	CB2A	SRA D	CB5A	BIT 3, D
CB03	RLC E	CB2B	SRA E	CB5B	BIT 3, E
CB04	RLC H	CB2C	SRA H	CB5C	BIT 3, H
CB05	RLC L	CB2D	SRA L	CB5D	BIT 3, L
CB06	RLC (HL)	CB2E	SRA (HL)	CB5E	BIT 3, (HL)
CB07	RLC A	CB2F	SRA A	CB5F	BIT 3, A
CB08	RRC B	CB38	SRL B	CB60	BIT 4, B
CB09	RRC C	CB39	SRL C	CB61	BIT 4, C
CB0A	RRC D	CB3A	SRL D	CB62	BIT 4, D
CB0B	RRC E	CB3B	SRL E	CB63	BIT 4, E
CB0C	RRC H	CB3C	SRL H	CB64	BIT 4, H
CB0D	RRC L	CB3D	SRL L	CB65	BIT 4, L
CB0E	RRC (HL)	CB3E	SRL (HL)	CB66	BIT 4, (HL)
CB0F	RRC A	CB3F	SRL A	CB67	BIT 4, A
CB10	RL B	CB40	BIT 0, B	CB68	BIT 5, B
CB11	RL C	CB41	BIT 0, C	CB69	BIT 5, C
CB12	RL D	CB42	BIT 0, D	CB6A	BIT 5, D
CB13	RL E	CB43	BIT 0, E	CB6B	BIT 5, E
CB14	RL H	CB44	BIT 0, H	CB6C	BIT 5, H
CB15	RL L	CB45	BIT 0, L	CB6D	BIT 5, L
CB16	RL (HL)	CB46	BIT 0, (HL)	CB6E	BIT 5, (HL)
CB17	RL A	CB47	BIT 0, A	CB6F	BIT 5, A
CB18	RR B	CB48	BIT 1, B	CB70	BIT 6, B
CB19	RR C	CB49	BIT 1, C	CB71	BIT 6, C
CB1A	RR D	CB4A	BIT 1, D	CB72	BIT 6, D
CB1B	RR E	CB4B	BIT 1, E	CB73	BIT 6, E

Table A-1 (cont'd.)

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
CB74	BIT 6, H	CB9C	RES 3, H	CBC4	SET 0, H
CB75	BIT 6, L	CB9D	RES 3, L	CBC5	SET 0, L
CB76	BIT 6, (HL)	CB0E	RES 3, (HL)	CBC6	SET 0, (HL)
CB77	BIT 6, A	CB9F	RES 3, A	CBC7	SET 0, A
CB78	BIT 7, B	CBA0	RES 4, B	CBC8	SET 1, B
CB79	BIT 7, C	CBA1	RES 4, C	CBC9	SET 1, C
CB7A	BIT 7, D	CBA2	RES 4, D	CBCA	SET 1, D
CB7B	BIT 7, E	CBA3	RES 4, E	CBCB	SET 1, E
CB7C	BIT 7, H	CBA4	RES 4, H	CBCC	SET 1, H
CB7D	BIT 7, L	CBA5	RES 4, L	CBCD	SET 1, L
CB7E	BIT 7, (HL)	CBA6	RES 4, (HL)	CBCE	SET 1, (HL)
CB7F	BIT 7, A	CBA7	RES 4, A	CBCF	SET 1, A
CB80	RES 0, B	CBA8	RES 5, B	CBD0	SET 2, B
CB81	RES 0, C	CBA9	RES 5, C	CBD1	SET 2, C
CB82	RES 0, D	CBAA	RES 5, D	CBD2	SET 2, D
CB83	RES 0, E	CBAB	RES 5, E	CBD3	SET 2, E
CB84	RES 0, H	CBAC	RES 5, H	CBD4	SET 2, H
CB85	RES 0, L	CBAD	RES 5, L	CBD5	SET 2, L
CB86	RES 0, (HL)	CBAE	RES 5, (HL)	CBD6	SET 2, (HL)
CB87	RES 0, A	CBAF	RES 5, A	CBD7	SET 2, A
CB88	RES 1, B	CBB0	RES 6, B	CBD8	SET 3, B
CB89	RES 1, C	CBB1	RES 6, C	CBD9	SET 3, C
CB8A	RES 1, D	CBB2	RES 6, D	CBDA	SET 3, D
CB8B	RES 1, E	CBB3	RES 6, E	CBDB	SET 3, E
CB8C	RES 1, H	CBB4	RES 6, H	CBDC	SET 3, H
CB8D	RES 1, L	CBB5	RES 6, L	CBDD	SET 3, L
CB8E	RES 1, (HL)	CBB6	RES 6, (HL)	CBDE	SET 3, (HL)
CB8F	RES 1, A	CBB7	RES 6, A	CBDF	SET 3, A
CB90	RES 2, B	CBB8	RES 7, B	CBE0	SET 4, B
CB91	RES 2, C	CBB9	RES 7, C	CBE1	SET 4, C
CB92	RES 2, D	CBBA	RES 7, D	CBE2	SET 4, D
CB93	RES 2, E	CBBB	RES 7, E	CBE3	SET 4, E
CB94	RES 2, H	CBBC	RES 7, H	CBE4	SET 4, H
CB95	RES 2, L	CBBD	RES 7, L	CBE5	SET 4, L
CB96	RES 2, (HL)	CBBE	RES 7, (HL)	CBE6	SET 4, (HL)
CB97	RES 2, A	CBBF	RES 7, A	CBE7	SET 4, A
CB98	RES 3, B	CBC0	SET 0, B	CBE8	SET 5, B
CB99	RES 3, C	CBC1	SET 0, C	CBE9	SET 5, C
CB9A	RES 3, D	CBC2	SET 0, D	CBEA	SET 5, D
CB9B	RES 3, E	CBC3	SET 0, E	CBEB	SET 5, E

Table A-1 (cont'd.)

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
CBEC	SET 5, H	CBF1	SET 6, C	CBF6	SET 6, (HL)
CBED	SET 5, L	CBF2	SET 6, D	CBF7	SET 6, A
CBEE	SET 5, (HL)	CBF3	SET 6, E	CBF8	SET 7, B
CBEF	SET 5, A	CBF4	SET 6, H	CBF9	SET 7, C
CBF0	SET 6, B	CBF5	SET 6, L	CBFA	SET 7, D

Table A-2 Z80 CPU INSTRUCTIONS SORTED BY MNEMONIC

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
8E	ADC A, (HL)	DD29	ADD IX, IX	CB49	BIT 1, C
DD8E05	ADC A, (IX + d)	DD39	ADD IX, SP	CB4A	BIT 1, D
FD8E05	ADC A, (IY + d)	FD09	ADD IY, BC	CB4B	BIT 1, E
8F	ADC A, A	FD19	ADD IY, DE	CB4C	BIT 1, H
88	ADC A, U	FD29	ADD IY, IY	CB4D	BIT 1, L
89	ADC A, C	FD39	ADD IY, SP	CB56	BIT 2, (HL)
8A	ADC A, D	A6	AND (HL)	DDCB0556	BIT 2, (IX + d)
8B	ADC A, E	DDA605	AND (IX + d)	FDCB0556	BIT 2, (IY + d)
8C	ADC A, H	FDA605	AND (IY + d)	CB57	BIT 2, A
8D	ADC A, L	A7	AND A	CB50	BIT 2, B
CE20	ADC A, N	A0	AND B	CB51	BIT 2, C
ED4A	ADC HL, BC	A1	AND C	CB52	BIT 2, D
ED5A	ADC HL, DE	A2	AND D	CB53	BIT 2, E
ED6A	ADC HL, HL	A3	AND E	CB54	BIT 2, H
ED7A	ADC HL, SP	A4	AND H	CB55	BIT 2, L
86	ADD A, (HL)	A5	AND L	CB5E	BIT 3, (HL)
DD8605	ADD A, (IX + d)	E620	AND N	DDCB055E	BIT 3, (IX + d)
FD8605	ADD A, (IY + d)	CB46	BIT 0, (HL)	FDCB055E	BIT 3, (IY + d)
87	ADD A, A	DDCB0546	BIT 0, (IX + d)	CB5F	BIT 3, A
80	ADD A, B	FDCB0546	BIT 0, (IY + d)	CB58	BIT 3, B
81	ADD A, C	CB47	BIT 0, A	CB59	BIT 3, C
82	ADD A, D	CB40	BIT 0, B	CB5A	BIT 3, D
83	ADD A, E	CB41	BIT 0, C	CB5B	BIT 3, E
84	ADD A, H	CB42	BIT 0, D	CB5C	BIT 3, H
85	ADD A, L	CB43	BIT 0, E	CB5D	BIT 3, L
C620	ADD A, N	CB44	BIT 0, H	CB66	BIT 4, (HL)
09	ADD HL, BC	CB45	BIT 0, L	DDCB0566	BIT 4, (IX + d)
19	ADD HL, DE	CB4E	BIT 1, (HL)	FDCB0566	BIT 4, (IY + d)
29	ADD HL, HL	DDCB054E	BIT 1, (IX + d)	CB67	BIT 4, A
39	ADD HL, SP	FDCB054E	BIT 1, (IY + d)	CB60	BIT 4, B
DD09	ADD IX, BC	CB4F	BIT 1, A	CB61	BIT 4, C
DD19	ADD IX, DE	BC48	BIT 1, B	CB62	BIT 4, D

Table A-2 (cont'd.)

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
CB63	BIT 4, E	BF	CP A	ED48	IN C, (C)
CB64	BIT 4, H	B8	CP B	ED50	IN D, (C)
CB65	BIT 4, L	B9	CP C	ED58	IN E, (C)
CB6E	BIT 5, (HL)	BA	CP D	ED60	IN H, (C)
DDCB056E	BIT 5, (IX + d)	BB	CP E	ED68	IN L, (C)
FDCB056E	BIT 5, (IY + d)	BC	CP H	34	INC (HL)
CB6F	BIT 5, A	BD	CP L	DD3405	INC (IX + d)
CB68	BIT 5, B	FE20	CP N	FD3405	INC (IY + d)
CB69	BIT 5, C	EDA9	CPD	3C	INC A
CB6A	BIT 5, D	ED89	CPDR	04	INC B
CB6B	BIT 5, E	EDA1	CPI	03	INC BC
CB6C	BIT 5, H	EDB1	CPIR	0C	INC C
CB6D	BIT 5, L	2F	CPL	14	INC D
CB76	BIT 6, (HL)	27	DAA	13	INC DE
DDCB0576	BIT 6, (IX + d)	35	DEC (HL)	1C	INC E
FDCB0578	BIT 5, (IY + d)	DD3505	DEC (IX + d)	24	INC H
CB77	BIT 6, A	FD3505	DEC (IY + d)	23	INC HL
CB70	BIT 6, B	3D	DEC A	DD23	INC IX
CB71	BIT 6, C	05	DEC B	FD23	INC IY
CB72	BIT 6, D	08	DEC BC	2C	INC L
CB73	BIT 6, E	0D	DEC C	33	INC SP
CB74	BIT 6, H	15	DEC D	EDAA	IND
CB75	BIT 6, L	1B	DEC DE	EDBA	INDR
CB7E	BIT 7, (HL)	1D	DEC E	EDA2	INI
DDCB057E	BIT 7, (IX + d)	25	DEC H	EDB2	INIR
FDCB057E	BIT 7, (IY + d)	2B	DEC HL	E9	JP (HL)
CB7F	BIT 7, A	DD28	DEC IX	DDE9	JP (IX)
CB78	BIT 7, B	FD2B	DEC IY	FDE9	JP (IY)
CB79	BIT 7, C	2D	DEC L	DA8405	JP C, NN
CB7A	BIT 7, D	3B	DEC SP	FA8405	JP M, NN
CB7B	BIT 7, E	F3	DI	D28405	JP NC, NN
CB7C	BIT 7, H	102E	DJNZ DIS	C38405	JP NN
CB7D	BIT 7, L	FB	EI	C28405	JP NZ, NN
DC8405	CALL C, NN	E3	EX (SP), HL	F28405	JP P, NN
FC8405	CALL M, NN	DDE3	EX (SP), IX	EA8405	JP PE, NN
D48405	CALL NC, NN	FDE3	EX (SP), IY	E28405	JP PO, NN
CD8405	CALL NN	08	EX AF, AF'	CA8405	JP Z, NN
C48405	CALL NZ, NN	EB	EX DE, HL	382E	JR C, DIS
F48405	CALL P, NN	D9	EXX	182E	JR DIS
EC8405	CALL PE, NN	76	HALT	302E	JR NC, DIS
E48405	CALL PO, NN	ED46	IM 0	202E	JR NZ, DIS
CC8405	CALL Z, NN	ED56	IM 1	282E	JR Z, DIS
3F	CCF	ED5E	IM 2	02	LD (BC), A
BE	CP (HL)	ED78	IN A, (C)	12	LD (DE), A
DDBE05	CP (IX + d)	DB20	IN A, (N)	77	LD (HL), A
FDBE06	CP (IY + d)	ED40	IN B, (C)	70	LD (HL), B

Table A-2 (cont'd.)

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
71	LD (HL), C	FD4605	LD B, (IY + d)	66	LD H, (HL)
72	LD (HL), D	47	LD B, A	DD6605	LD H, (IX + d)
73	LD (HL), E	40	LD B, B	FD6606	LD H, (IY + d)
74	LD (HL), H	41	LD B, C	67	LD H, A
75	LD (HL), L	42	LD B, D	60	LD H, B
3620	LD (HL), N	43	LD B, E	61	LD H, C
DD7705	LD (IX + d), A	44	LD B, H, NN	62	LD H, D
DD7005	LD (IX + d), B	45	LD B, L	63	LD H, E
DD7105	LD (IX + d), C	0620	LD B, N	64	LD H, H
DD7205	LD (IX + d), D	ED4B8405	LD BC, (NN)	65	LD H, L
DD7305	LD (IX + d), E	018405	LD BC, NN	2620	LDH, N
DD7405	LD (IX + d), H	4E	LD C, (HL)	2A8405	LD HL, (NN)
DD7505	LD (IX + d), L	DD4E05	LD C, (IX + d)	218405	LD HL, NN
DD360520	LD (IX + d), N	FD4E05	LD C, (IY + d)	ED47	LD I, A
FD7705	LD (IY + d), A	4F	LD C, A	DD2AB405	LD IX, (NN)
FD7005	LD (IY + d), B	48	LD C, B	DD218405	LD IX, NN
FD7105	LD (IY + d), C	49	LD C, C	FD2A8405	LD IY, (NN)
FD7205	LD (IY + d), D	4A	LD C, D	FD218405	LD IY, NN
FD7305	LD (IY + d), E	4B	LD C, E	6E	LD L, (HL)
FD7405	LD (IY + d), H	4C	LD C, H	DD6E05	LD L, (IX + d)
FD7505	LD (IY + d), L	4D	LD C, L	FD6E05	LD L, (IY + d)
FD360520	LD (IY + d), N	0E20	LD C, N	6F	LD L, A
328405	LD (NN), A	56	LD D, (HL)	68	LD L, B
ED438405	LD (NN), BC	DD5605	LD D, (IX + d)	69	LD L, C
ED538405	LD (NN), DE	FD5605	LD D, (IY + d)	6A	LD L, D
228405	LD (NN), HL	57	LD D, A	6B	LD L, E
DD228405	LD (NN), IX	50	LD D, B	6C	LD L, H
FD228405	LD (NN), IY	51	LD D, C	6D	LD L, L
ED738405	LD (NN), SP	52	LD D, D	2E20	LD L, N
0A	LD A, (BC)	53	LD D, E	ED788405	LD SP, (NN)
1A	LD A, (DE)	54	LD D, H	F9	LD SP, HL
7E	LD A, (HL)	55	LD D, L	DDF9	LD SP, IX
DD7E05	LD A, (IX + d)	1620	LD D, N	FDF9	LD SP, IY
FD7E05	LD A, (IY + d)	ED5B8405	LD DE, (NN)	318405	LD SP, NN
3A8405	LD A, (NN)	118405	LD DE, NN	EDA8	LDD
7F	LD A, A	5E	LD E, (HL)	ED88	LDDR
78	LD A, B	DD5E05	LD E, (IX + d)	EDA0	LDI
79	LD A, C	FD5E05	LD E, (IY + d)	EDB0	LDIR
7A	LD A, D	5F	LD E, A	ED44	NEG
7B	LD A, E	58	LD E, B	00	NOP
7C	LD A, H	59	LD E, C	B6	OR (HL)
ED57	LD A, I	5A	LD E, D	DDB605	OR (IX + d)
7D	LD A, L	5B	LD E, E	FDB605	OR (IY + d)
3E20	LD A, N	5C	LD E, H	B7	OR A
46	LD B, (HL)	5D	LD E, L	B0	OR B
DD4605	LD B, (IX + d)	1E20	LD E, N	B1	OR C

Table A-2 (cont'd.)

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
B2	OR D	CB8B	RES 1, E	CBB7	RES 6, A
B3	OR E	CB8C	RES 1, H	CBB0	RES 6, B
B4	OR H	CB8D	RES 1, L	CBB1	RES 6, C
B5	OR L	CB96	RES 2, (HL)	CBB2	RES 6, D
F620	OR N	DDCB0596	RES 2, (IX + d)	CBB3	RES 6, E
EDBB	OTDR	FDCB0596	RES 2, (IY + d)	CBB4	RES 6, H
EDB3	OTIR	CB97	RES 2, A	CBB5	RES 6, L
ED79	OUT (C), A	CB90	RES 2, B	CBBE	RES 7, (HL)
ED41	OUT (C), B	CB91	RES 2, C	DDCB05BE	RES 7, (IX + d)
ED49	OUT (C), C	CB92	RES 2, D	FDCB05BE	RES 7, (IX + d)
ED51	OUT (C), D	CB93	RES 2, E	CBBF	RES 7, A
ED59	OUT (C), E	CB94	RES 2, H	CBB8	RES 7, B
ED61	OUT (C), H	CB95	RES 2, L	CBB9	RES 7, C
ED69	OUT (C), L	CB9E	RES 3, (HL)	CBBA	RES 7, D
D320	OUT (N), A	DDCB059E	RES 3, (IX + d)	CBBB	RES 7, E
EDAB	OUTD	FDCB059E	RES 3, (IY + d)	CBBC	RES 7, H
EDA3	OUTI	CB9F	RES 3, A	CBBD	RES 7, L
F1	POP AF	CB98	RES 3, B	C9	RET
C1	POP BC	CB99	RES 3, C	D8	RET C
D1	POP DE	CB9A	RES 3, D	F8	RET M
E1	POP HL	CB9B	RES 3, E	D0	RET NC
DDE1	POP IX	CB9C	RES 3, H	C0	RET NZ
FDE1	POP IY	CB9D	RES 3, L	F0	RET P
F5	PUSH AF	CBA6	RES 4, (HL)	E8	RET PE
C5	PUSH BC	DDCB05AB	RES 4, (IX + d)	E0	RET PO
D5	PUSH DE	FDCB05AB	RES 4, (IY + d)	C8	RET Z
E5	PUSH HL	CBA7	RES 4, A	ED4D	RETI
DDE5	PUSH IX	CBA0	RES 4, B	ED45	RETN
FDE5	PUSH IY	CBA1	RES 4, C	CB16	RL (HL)
CB86	RES 0, (HL)	CBA2	RES 4, D	DDCB0516	RL (IX + d)
DDCB0586	RES 0, (IX + d)	CBA3	RES 4, E	FDCB0516	RL (IY + d)
FDCB0586	RES 0, (IY + d)	CBA4	RES 4, H	CB17	RL A
CB87	RES 0, A	CBA5	RES 4, L	CB10	RL B
CB80	RES 0, B	CBAE	RES 5, (HL)	CB11	RL C
CB81	RES 0, C	DDCB05AE	RES 5, (IX + d)	CB12	RL D
CB82	RES 0, D	FDCB05AE	RES 5, (IY + d)	CB13	RL E
CB83	RES 0, E	CBAF	RES 5, A	CB14	RL H
CB84	RES 0, H	CBA8	RES 5, B	CB15	RL L
CB85	RES 0, L	CBA9	RES 5, C	17	RLA
CB8E	RES 1, (HL)	CBAA	RES 5, D	CB06	RLC (HL)
DDCB058E	RES 1, (IX + d)	CBAB	RES 5, E	DDCB0506	RLC (IX + d)
FDCB058E	RES 1, (IY + d)	CBAC	RES 5, H	FDCB0506	RLC (IY + d)
CB8F	RES 1, A	CBAD	RES 5, L	CB07	RLC A
CB88	RES 1, B	CBB6	RES 6, (HL)	CB00	RLC B
CB89	RES 1, C	DDCB05B6	RES 6, (IX + d)	CB01	RLC C
CB8A	RES 1, D	FDCB05B6	RES 6, (IY + d)	CB02	RLC D

Table A-2 (cont'd.)

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
CB03	RLC E	DE20	SBC A, N	CBE6	SET 4, (HL)
CB04	RLC H	ED42	SBC HL, BC	DDCB05E6	SET 4, (IX + d)
CB05	RLC L	ED52	SBC HL, DE	FDCB05E6	SET 4, (IY + d)
07	RLCA	ED62	SBC HL, HL	CBE7	SET 4, A
ED6F	RLD	ED72	SBC HL, SP	CBE0	SET 4, B
CB1E	RR (HL)	37	SCF	CBE1	SET 4, C
DDCB051E	RR (IX + d)	CBC6	SET 0, (HL)	CBE2	SET 4, D
FDCB051E	RR (IY + d)	DDCB05CS	SET 0, (IX + d)	CBE3	SET 4, E
CB1F	RR A	FDCB05C6	SET 0, (IY + d)	CBE4	SET 4, H
CB18	RR B	CBC7	SET 0, A	CBE5	SET 4, L
CB19	RR C	CBC0	SET 0, B	CBEE	SET 5, (HL)
CB1A	RR D	CBC1	SET 0, C	DDCB05EE	SET 5, (IX + d)
CB1B	RR E	CBC2	SET 0, D	FDCB05EE	SET 5, (IY + d)
CB1C	RR H	CBC3	SET 0, E	CBEF	SET 5, A
CB1D	RR L	CBC4	SET 0, H	CBE8	SET 5, B
1F	RRA	CBC5	SET 0, L	CBE9	SET 5, C
CB0E	RRC (HL)	CBCE	SET 1, (HL)	CBEA	SET 5, D
DDCB050E	RRC (IX + d)	DDCB05CE	SET 1, (IX + d)	CBEB	SET 5, E
FDCB050E	RRC (IY + d)	FDCB05CE	SET 1, (IY + d)	CBEC	SET 5, H
CB0F	RRC A	CBCF	SET 1, A	CBED	SET 5, L
CB08	RRC B	CBC8	SET 1, B	CBF6	SET 6, (HL)
CB09	RRC C	CBC9	SET 1, C	DDCB05F6	SET 6, (IX + d)
CB0A	RRC D	CBCA	SET 1, D	FDCB05F6	SET 6, (IY + d)
CB0B	RRC E	CBCB	SET 1, E	CBF7	SET 6, A
CB0C	RRC H	CBCC	SET 1, H	CBF0	SET 6, B
CB0D	RRC L	CBCD	SET 1, L	CBF1	SET 6, C
0F	RRCA	CBD6	SET 2, (HL)	CBF2	SET 6, D
ED67	RRD	DDCB05D6	SET 2, (IX + d)	CBF3	SET 6, E
C7	RST 0	FDCB05D6	SET 2, (IY + d)	CBF4	SET 6, H
D7	RST 10H	CBD7	SET 2, A	CBF5	SET 6, L
DF	RST 18H	CBD0	SET 2, B	CBFE	SET 7, (HL)
E7	RST 20H	CBD1	SET 2, C	DDCB05FE	SET 7, (IX + d)
EF	RST 28H	CBD2	SET 2, D	FDCB05FE	SET 7, (IY + d)
F7	RST 30H	CBD3	SET 2, E	CBFF	SET 7, A
FF	RST 38H	CBD4	SET 2, H	CBF8	SET 7, B
CF	RST 8	CBD5	SET 2, L	CBF9	SET 7, C
9E	SBC A, (HL)	CBD8	SET 3, B	CBFA	SET 7, D
DD9E05	SBC A, (IX + d)	CBDE	SET 3, (HL)	CBFB	SET 7, E
FD9E05	SBC A, (IY + d)	DDCB05DE	SET 3, (IX + d)	CBFC	SET 7, H
9F	SBC A, A	FDCB05DE	SET 3, (IY + d)	CBFD	SET 7, L
98	SBC A, B	CBDF	SET 3, A	CB26	SLA (HL)
99	SBC A, C	CBD9	SET 3, C	DDCB0526	SLA (IX + d)
9A	SBC A, D	CBDA	SET 3, D	FDCB0526	SLA (IY + d)
9B	SBC A, E	CBDB	SET 3, E	CB27	SLA A
9C	SBC A, H	CBDC	SET 3, H	CB20	SLA B
9D	SBC A, L	CBDD	SET 3, L	CB21	SLA C

Table A-2 (cont'd.)

OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT	OBJ CODE	SOURCE STATEMENT
CB22	SLA D	FDCB053E	SRL (IY + d)	93	SUB E
CB23	SLA E	CB3F	SRL A	94	SUB H
CB24	SLA H	CB38	SRL B	95	SUB L
CB25	SLA L	CB39	SRL C	D620	SUB N
CB2E	SRA (HL)	CB3A	SRL D	AE	XOR (HL)
DDCB052E	SRA (IX + d)	CB3B	SRL E	DDAE05	XOR (IX + d)
FDCB052E	SRA (IY + d)	CB3C	SRL H	FDAE05	XOR (IY + d)
CB2F	SRA A	CB3D	SRL L	AF	XOR A
CB28	SRA B	96	SUB (HL)	A8	XOR B
CB29	SRA C	DD9605	SUB (IX + d)	A9	XOR C
CB2A	SRA D	FD9605	SUB (IY + d)	AA	XOR D
CB2B	SRA E	97	SUB A	AB	XOR E
CB2C	SRA H	90	SUB B	AC	XOR H
CB2D	SRA L	91	SUB C	AD	XOR L
CB3E	SRL (HL)	92	SUB D	EE20	XOR N
DDCB053E	SRL (IX + d)				

Table A-3 ASCII CODE

HEX CODE	MEANING	COMMENTS
00	NUL	null
01	SOH	start of heading
02	STX	start text
03	ETX	end text
04	EOT	end of transmission
05	ENQ	enquiry
06	ACK	acknowledgment
07	BEL	bell
08	BS	back space
09	HT	horizontal tab
0A	LF	line feed
0B	VT	vertical tab
0C	FF	form feed
0D	CR	carriage return
0E	SO	shift out
0F	SI	shift in
10	DLE	data link escape
11	DC1	direct control 1
12	DC2	direct control 2
13	DC3	direct control 3
14	DC4	direct control 4

Table A-3 (cont'd.)

CODE	MEANING	COMMENTS
15	NAK	negative acknowledgment
16	SYN	synchronous idle
17	ETB	end of transmission block
18	CAN	cancel
19	EM	end of medium
1A	SUB	substitute
1B	ESC	escape
1C	FS	form separator
1D	GS	group separator
1E	RS	record separator
1F	US	unit separator
20	(special)	—
21	!	—
22	"	—
23	#	—
24	$	—
25	%	—
26	&	—
27	'	—
28	(	—
29	)	—
2A	*	—
2B	+	—
2C	,	—
2D	-	—
2E	.	—
2F	/	—
30	0	—
31	1	—
32	2	—
33	3	—
34	4	—
35	5	—
36	6	—
37	7	—
38	8	—
39	9	—
3A	:	—
3B	;	—
3C	>	—

Table A-3 (cont'd.)

HEX CODE	MEANING	COMMENTS
3D	=	–
3E	<	–
3F	?	–
40	@	–
41	A	–
42	B	–
43	C	–
44	D	–
45	E	–
46	F	–
47	G	–
48	H	–
49	I	–
4A	J	–
4B	K	–
4C	L	–
4D	M	–
4E	N	–
4F	O	–
50	P	–
51	Q	–
52	R	–
53	S	–
54	T	–
55	U	–
56	V	–
57	W	–
58	X	–
59	Y	–
5A	Z	–
5B	[	–
5C	/	–
5D	]	–
5E	·	–
5F	|	–
60	Λ	–
61	a	–
62	b	–
63	c	–
64	d	–

Table A-3 (cont'd.)

HEX CODE	MEANING	COMMENTS
65	e	—
66	f	—
67	g	—
68	h	—
69	i	—
6A	j	—
6B	k	—
6C	l	—
6D	m	—
6E	n	—
6F	o	—
70	p	—
71	q	—
72	r	—
73	s	—
74	t	—
75	u	—
76	v	—
77	w	—
78	x	—
79	y	—
7A	z	—
7B	{	—
7C	|	—
7D	}	—
7E	~	—
7F	DEL	—

BAUDOT TELETYPEWRITER CODE

This five-bit code is obsolete and is not in general use in newly designed equipment. But there is a considerable amount of older equipment on the market and still in use. Many surplus commercial and military teletypewriters are used by amateur computer enthusiasts, and they also remain in service in older systems. The Teletype Corporation still offers Baudot coding on their new machines as an option for owners of older systems.

Table A-4 BAUDOT CODE

B5	B4	B3	B2	B1	REGULAR	SHIFTED
0	0	0	0	0	BLANK	BLANK
0	0	0	0	1	E	3
0	0	0	1	0	linefeed	linefeed
0	0	0	1	1	A	-
0	0	1	0	0	space	space
0	0	1	0	1	S	Bell
0	0	1	1	0	I	8
0	0	1	1	1	U	7
0	1	0	0	0	Car. Ret.	Car. Ret.
0	1	0	0	1	D	$
0	1	0	1	0	R	4
0	1	0	1	1	J	'
0	1	1	0	0	N	,
0	1	1	0	1	F	!
0	1	1	1	0	C	:
0	1	1	1	1	K	(
1	0	0	0	0	T	5
1	0	0	0	1	Z	"
1	0	0	1	0	L	)
1	0	0	1	1	W	2
1	0	1	0	0	H	#
1	0	1	0	1	Y	6
1	0	1	1	0	P	0
1	0	1	1	1	Q	1
1	1	0	0	0	O	9
1	1	0	0	1	B	?
1	1	0	1	0	G	&
1	1	0	1	1	(figures)	(figures)
1	1	1	0	0	M	.
1	1	1	0	1	X	/
1	1	1	1	0	V	;
1	1	1	1	1	(letters)	(letters)

EBCDIC CODE

This code is used by IBM on some of their equipment. It must be considered when one is interfacing IBM systems, IBM-compatible equipment, or trying to use surplus IBM Selectric typewriters (not all of which used EBCDIC; some also used BCD, while others used correspondence code).

Table A-5 EBCDIC CODE

CHARACTER	HEX CODE	CHARACTER	HEX CODE
A	C1	K	D2
B	C2	L	D3
C	C3	M	D4
D	C4	N	D5
E	C5	O	D6
F	C6	P	D7
G	C7	Q	D8
H	C8	R	D9
I	C9	S	E2
J	D1	T	E3
U	E4	4	4
V	E5	5	5
W	E6	6	6
X	E7	7	7
Y	E8	8	8
Z	E9	9	9
0	0		
1	1		
2	2		
3	3		

APPENDIX B
Z80 FAMILY SUPPORT CHIPS

The Z80 is not a single-chip computer. In order to make the Z80 microprocessor chip think that it is a real live computer we need additional, external circuitry. In some commercial products, this external circuitry takes the form of TTL and/or CMOS devices connected to perform the desired function. But Zilog, Inc. makes it easier to make a computer by using certain external special-function integrated circuits.

Two of the special-function devices are used to provide serial and parallel input/output capability. The Z80-SIO device is a serial I/O chip, while the Z80-PIO is a parallel I/O port. These devices are second-sourced by Mostek under the type numbers MK3884 (Z80-SIO) and MK3881 (Z80-PIO).

There is also a direct memory access device called the Z80-DMA (Mostek MK3883). Direct memory access in a computer allows the external memory to be written to, or read from, by a peripheral device without first going through the CPU. This allows the operation to be performed much more rapidly, and is conservative of CPU time—a precious commodity in some applications.

The Z80-CTC (Mostek MK3882) is a four-channel, multimode counter/timer circuit. It provides counter and timer capability in Z80-based microcomputer systems.

Z80-PIO

The Zilog-PIO (Mostek MK3881) is used as a parallel I/O port controller. It contains two ports, and is user programmable. The Z80-PIO contains two completely independent, eight-bit, bidirectional ports. Complete handshaking capability is permitted, so the device can be used for synchronous transfers.

The Z80-PIO can be programmed to operate in four different modes: *byte output, byte input, byte bidirectional bus* (port A only), and *bit control.*

The byte output mode, also called mode-Ø, is used to allow the CPU to write data to the peripheral via the CPU data bus. If mode-Ø is selected, a *data write* operation causes a handshake signal (*ready*) to be generated. This signal is used to let the peripheral know that the data are available and valid. Note that the data remain available, and the *ready* signal remains HIGH, until a strobe is received back from the peripheral.

The *byte input mode,* also called mode-1, allows the selected port to behave as an input port only. When a *data read* operation is performed by the CPU, the PIO will issue a *ready* signal to the peripheral. This tells the peripheral that the Z80 CPU is now in a condition to receive the input data. The peripheral responds by issuing a strobe that causes the data to be transferred to the data input register of the PIO.

The *byte bidirectional mode,* also called mode-2, uses the port as a bidirectional, eight-bit, I/O port. Mode-2 uses all four possible handshake lines. Because of this restriction, only port A can be used in the bidirectional mode.

The *bit control mode,* also called mode-3, is used for status and control applications. Mode-3 does not make use of the handshake signals. This mode is used to define which port data bus lines will be inputs and which will be outputs. The next word fed to the PIO after mode-3 is selected must define these conditions.

Figure B-1 shows the pin-outs for the Z80-PIO, while below are the definitions of the different types of pins.

DØ-D7 — These pins connect to the Z80 CPU data bus, and are both bidirectional and tri-state. All command signals and data passed between the CPU and the PIO, in either direction, must be passed over these lines.

B/A SEL — This active-HIGH input will select either port A or port B. A LOW on B/A SEL will select port A, whereas a HIGH will select port A.

C/D SEL — This active-HIGH input selects the type of data transfer to take place between the CPU and PIO. A LOW on this line tells the PIO that the data on the Z80 data bus are I/O

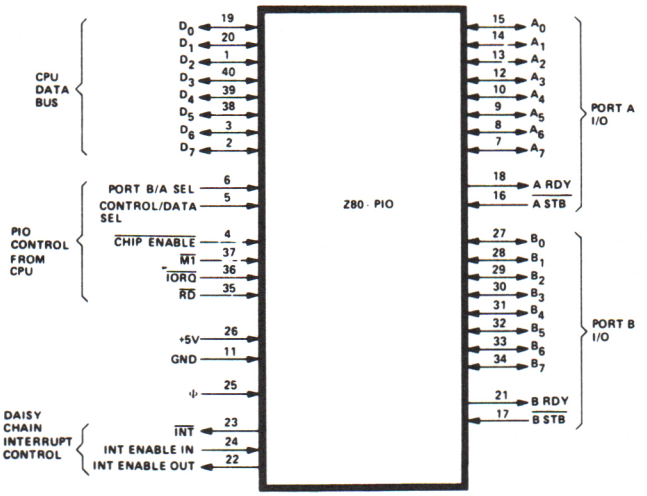


FIGURE B-1. Z80-PIO pin-outs.

data. But a HIGH will tell the PIO that the data being transferred are a command for the port selected by B/A SEL.

$\overline{CE}$ Active-LOW input that acts as a chip enable. A LOW on this terminal allows the PIO to accept command/data inputs from the Z80 CPU during any write cycle, or to send data to the Z80 CPU during any read cycle.

$\overline{M1}$ This terminal synchronizes the PIO to the CPU, and is generally connected to the similarly named terminal on the CPU chip. Indicates that an M1 machine cycle is in progress.

$\overline{IORQ}$ Input/output request line from the Z80 CPU chip that is part of the sync system. Usually connected to the similarly named terminal on the Z80 device.

$\overline{RD}$ Active-LOW input that detects the read cycle of the Z80.

IEI Interrupt Enable Input. This is an active-HIGH input.

$A_{\emptyset}$-A7 Tri-state, bidirectional address bus for port A.

$\overline{A\ STB}$ Active-LOW input that strobes port A from peripheral device.

A RDY Active-HIGH output signals that the A register is ready.

$B_{\emptyset}$-B7 Tri-state, bidirectional, address bus for port B.

$\overline{B\ STB}$ Active-LOW input that allows peripheral device to strobe port B.

B RDY Active-HIGH output that signals that the B register is ready.

Z80-SIO

The Z80-SIO device is a serial I/O chip that interfaces directly with the Z80 CPU chip. It is similar to the Z80-PIO in that it is a programmable two-channel device. The SIO, however, transmits the data in the *serial* stream, i.e., one bit at a time. Parallel transfer is, of course, faster in most cases. But often a serial transfer is preferred because it reduces the hardware overhead between the computer and the peripheral with which it is communicating. Even when the "run" is only a short distance, it is often much less costly to use a serial data transfer because only one pair of wires, one telephone line, or one radio communications channel is required. The Z80-SIO is designed to handle just about any reasonable serial bit protocol. Like the other chips of the Z80 family, it is operated from a single +5-volt DC supply and uses only a single-phase clock.

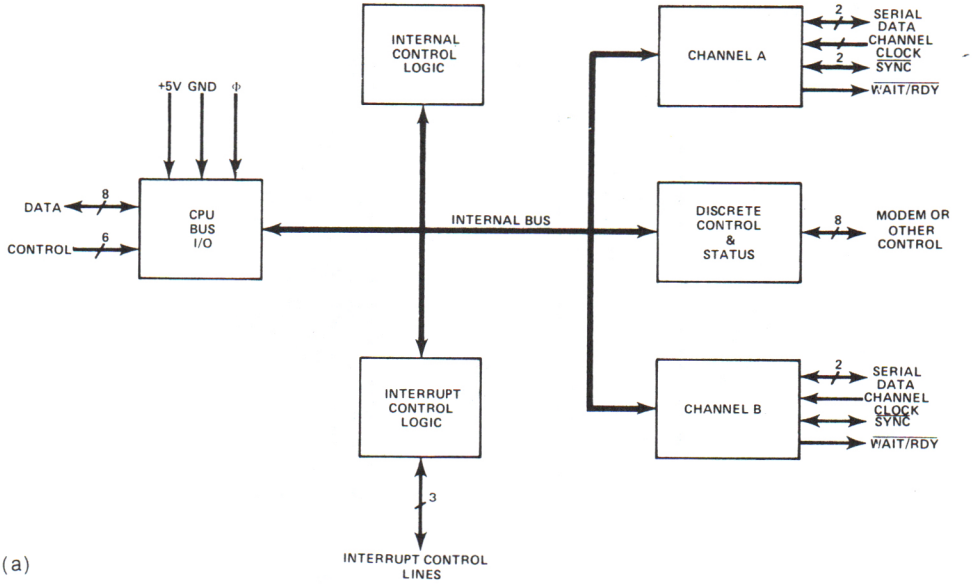
The two channels (also labeled A and B, as in the PIO device) are totally independent of each other, except for power supply and CPU bus connections. The SIO channels are full-duplex, so data can be transmitted and received simultaneously. The Z80-SIO allows data rates from zero to 550,000 bits per second.

Both receiver and transmitter registers are fully buffered. But in the case of the transmitter section, the registers are doubly buffered. The receiver registers, on the other hand, are quadruply buffered.

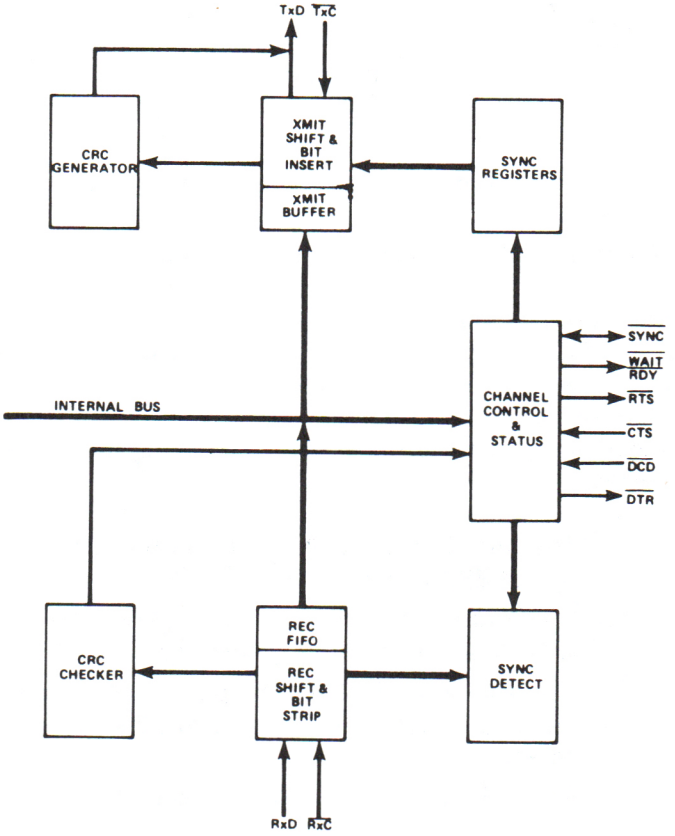
The Z80-SIO is capable of *asynchronous* operation (in which it behaves much like an ordinary UART, but with a Z80-system flavor), *synchronous binary* operation, and HDLC/IBM-SDLC operation. The SIO provides eight modem control inputs/outputs, allows daisy chain priority interrupt logic to automatically provide the vector word, and permits both CRC-16 and CRC-CCIT (—0/—1).

The SIO looks very much like the ordinary UART in its asynchronous mode. It can be programmed for 5, 6, 7, and 8 eight-bit words. Like the UART, it will provide 1, 1.5, or 2 stop bits at the end of each transmitted word. The CPU, incidentally, need not provide these bits; the SIO adds them to the word received from the CPU before the word is transmitted. Also like the UART, the SIO will provide parity bits (even, odd, none), and detection of parity, framing errors, and overrun. Unlike most UARTs, however, the SIO also provids for the generation and detection of breaks. Clock rates of 1X, 16X, 32X, and 64X the data rate are permitted.

Figure B-2 shows the organization of the Z80-PIO device. In Fig. B-2(a) we see the overall block diagram of the device, while Fig. B-2(b) shows the



(a)



(b)

FIGURE B-2. Z80-PIO organization. (a) Overall internal block diagram; (b) Channel block diagram.

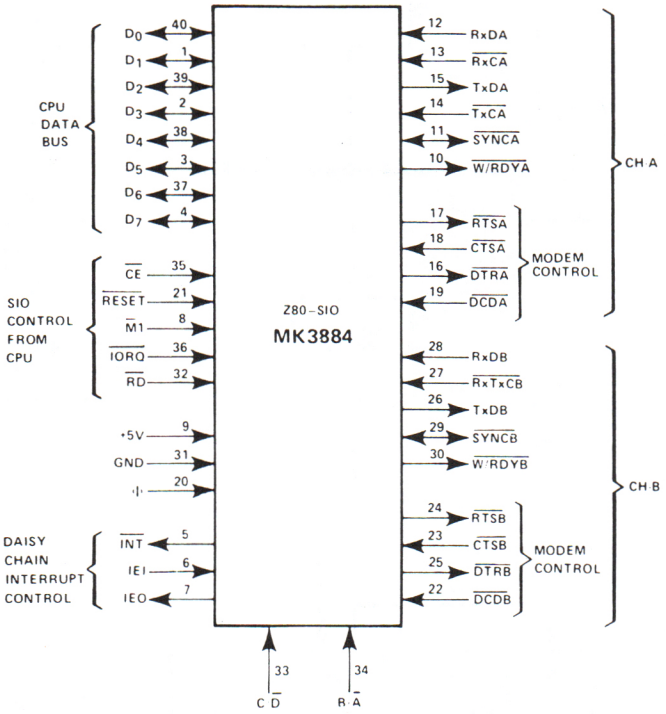


FIGURE B-3. Z80-SIO pin-outs.

block diagram for the channels. The input section from the CPU receives eight data bus lines, and six control signal lines. Once inside, the device operates from an internal bus not accessible to the outside world. There are two sections for channels A and B, some internal control logic, the interrupt section, and a discrete control section (used with modems and other controlling devices).

The pin-outs for the Z80-SIO are shown in Fig. B-3, and are listed in detail below:

DØ-D7	Tri-state, bidirectional data bus to/from Z80-CPU and rest of Z80 system.
B/A	Channel A/B select. Channel A is selected when this pin is LOW, and channel B is selected when it is HIGH.
C/D	Control/data select. If this input is HIGH, then the control mode is selected, but if it is LOW, then the data mode is selected.
$\overline{M1}$	Active-LOW input that detects the M1 machine cycle in Z80.

$\overline{\text{IORQ}}$	Active-LOW input that detects the Input/Output ReQuest state of the Z80 CPU.
$\overline{\text{RD}}$	Active-LOW input that detects the read cycle of the Z80 CPU.
Ø	Clock terminal.
$\overline{\text{RESET}}$	Active-LOW input that resets the system. Placing a LOW on this terminal has the following results: both receivers and transmitters, are disabled, TDA/TDB are forced marking, modem controls are forced HIGH, and all interrupts are disabled. *Note:* The control registers of the SIO *must* be rewritten from the CPU before the SIO can be again used.
IEI	Active-HIGH interrupt enable input.
IEO	Active-HIGH output. Note that IEI/IEO are used together to form a daisy-chain priority interrupt control function.
$\overline{\text{INT}}$	Active-LOW output to the interrupt request line of the Z80. Note that this terminal is an open-drain type.
WAIT/ READY A WAIT/ READY B	These lines, one for each channel, have two principal functions. In one case, they can be used as ready lines for the Z80-DMA (direct memory access) controller. In another, they can be used to synchronize the Z80 CPU to the Z80-SIO (i.e., to sync the data rate between SPU and SIO).
$\overline{\text{CTSA}}$ $\overline{\text{CTSB}}$	These lines, one for each channel, provide a *clear to send* function. Both are active-LOW inputs. If programmed for *auto enable,* then these pins will act as transmitter enable controls. But when not programmed for auto enable, they can be programmed for general control purposes. *Note:* These pins are buffered through Schmitt-trigger circuits, thereby allowing slow rise-time signals.
$\overline{\text{DCDA}}$ $\overline{\text{DCDB}}$	Data Carrier Detect. These two active-LOW inputs serve as receiver enable control signals.
RDA/ RDB	Active-HIGH receiver data inputs.
TDA/TDB	Active-HIGH transmit data outputs.
$\overline{\text{RCA}}$/ $\overline{\text{RCB}}$	Schmitt-trigger buffered, active-LOW receiver clock inputs.
TAC/TCB	Same as above, but transmitter clocks.
$\overline{\text{RTSA}}$/ $\overline{\text{RTSB}}$	Active-LOW outputs providing *request-to-send* signals.

$\overline{\text{DTRA}}$/ $\overline{\text{DTRB}}$	Active-LOW outputs providing *data-terminal-ready* signals.
$\overline{\text{SYNCA}}$/ $\overline{\text{SYNCB}}$	Used for synchronization of external characters.

Z80-DMA

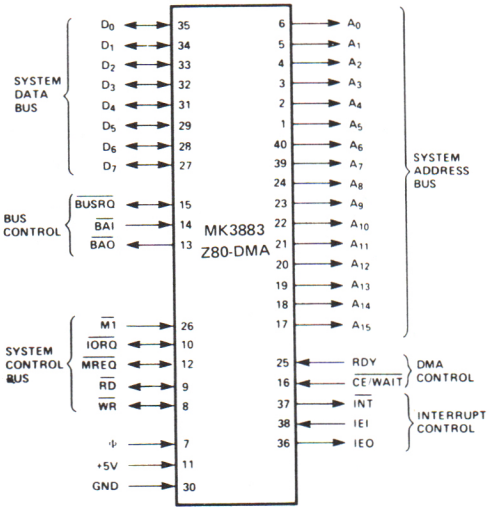
The Z80-DMA (Mostek MK3883) is a *direct memory access* controller. This type of operation is very useful in a computer, because it speeds up direct transfers between an external device, or peripheral, and the memory because it allows bypassing of the CPU. Ordinarily, if you wanted to transfer a data word from some peripheral device and a specific memory location, you would have to execute an input instruction to move the data into the accumulator first. Then a second instruction would be required in order to move the data from the CPU to the desired memory location. Unless the data are to be used immediately after input, then such would be a waste of valuable time. DMA allows the data to be placed directly into the desired location from the peripheral.

The DMA chip allows three modes, or classes, of operation: *transfer only, search only,* and *search-transfer.* There are also four types of operation: *single byte at a time, continuous burst* (as long as ports are ready), *continuous* (CPU locked out), and *transparent* (i.e., it steals time from refresh cycles).

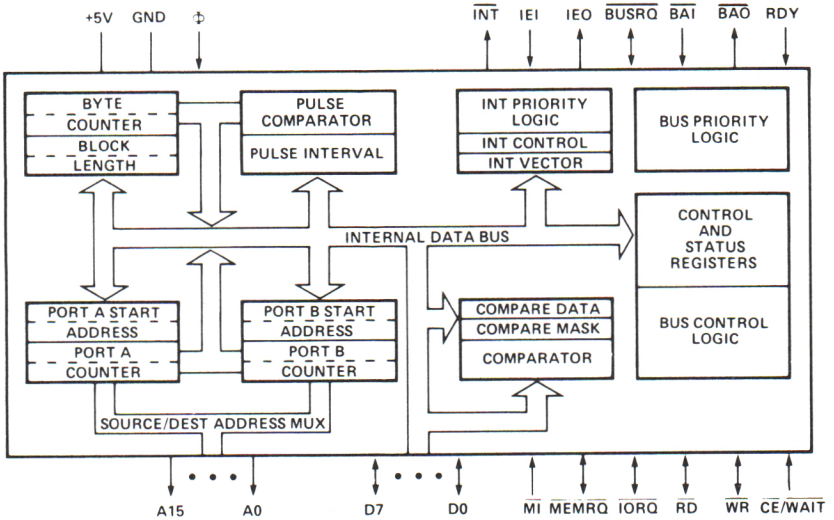
Three types of interrupt are allowed. In one case, it will interrupt the CPU only when a match to a desired word is found. It will also interrupt on *end-of-block* or *ready.* The DMA can be enabled, disabled, or reset totally under software control.

Figure B-4(a) shows the pin-outs for the Z80-DMA, while Fig. B-4(b) shows the internal block diagram. The pin-out functions are discussed below:

$A_{\emptyset}$-A15	System address bus (from Z80 and memory). This sixteen-bit address bus can, like the Z80 bus, address all 64K of allowed memory.
$D_{\emptyset}$-D7	Data bus from CPU and memory. These tri-state input/output pins carry three types of data: commands from the Z80 CPU, DMA status (from memory/peripherals), and data from the memory/peripherals.
ϕ	System clock.
$\overline{\text{M1}}$	Active-LOW input detects the M1 machine cycle in the Z80 CPU.
$\overline{\text{IORQ}}$	Used as an input/output request to/from the CPU bus.
$\overline{\text{MREQ}}$	Used as a memory request to/from Z80 system bus.



(a)



(b)

FIGURE B-4. (a) Z80-DMA pin-outs; (b) Z80-DMA block diagram.

$\overline{RD}$	Read to/from Z80 CPU bus.
$\overline{WR}$	Write signal to/from Z80 CPU bus.
$\overline{CE/WAIT}$	May be used as either chip enable or $\overline{WAIT}$.
$\overline{BUSRQ}$	Bus request is used to request control of the data bus from the Z80 CPU.

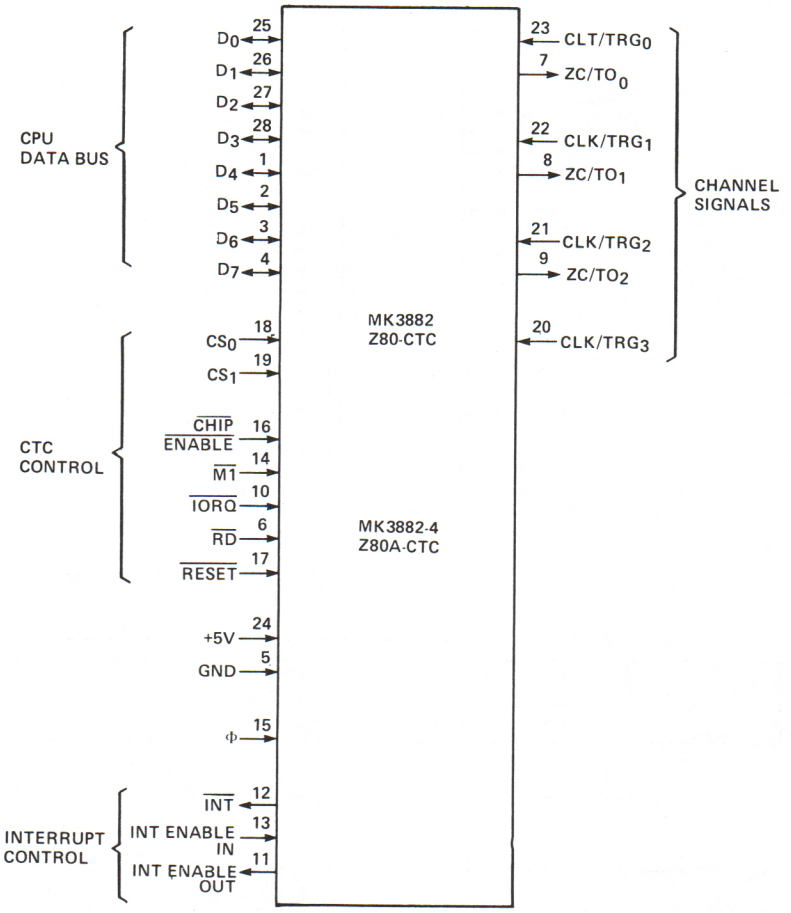


FIGURE B-5. Z80-CTC pin-outs.

$\overline{\text{BAI}}$	Input that tells the Z80-DMA that the CPU has granted it control of the bus. It is a bus acknowledge input.
$\overline{\text{BAO}}$	Bus acknowledge output that allows daisy chain connection of DMA-requesting peripherals.
$\overline{\text{INT}}$	Active-LOW output that tells that Z80 CPU that an interrupt is requested.
IEI	Active-HIGH interrupt enable input.
IEO	Active-HIGH interrupt enable output. Forms ability to daisy chain, when used in conjunction with IEI.
RDY	Active-HIGH/LOW (i.e., programmable) input that tells the Z80-DMA when a peripheral device is ready for a write/read operation.

Z80-CTC

The Z80-CTC (Mostek MK3882) is a universal counter-timer chip that can provide all of the counter/timer requirements for a Z80-based computer. There are four independent channels in the Z80-CTC. Consistent with the design of the rest of the Z80 family, this device requires only a single +5-volt DC power supply and a single-phase clock. Each of the four channels can operate either a counter or a timer.

The Z80-CTC pin-outs are shown in Fig. B-5, and their respective descriptions are given below:

DØ-D7	Bidirectional tri-state data bus to/from CPU.
CSØ-CS1	Active-HIGH channel select inputs.
$\overline{\text{CE}}$	Active-LOW chip enable input.
Ø	System clock.
$\overline{\text{M1}}$	Active-LOW input from CPU that detects the M1 machine cycle.
$\overline{\text{IORQ}}$	Active-LOW input that detects the input/output request state of the CPU.
$\overline{\text{RD}}$	Active-LOW input that detects the Z80 CPU read cycle.
IEI	Active-HIGH interrupt enable input.
IEO	Active-HIGH interrupt enable output. Used with IEI to permit daisy chaining.
$\overline{\text{INT}}$	Active-LOW, open-drain, output to the Z80 CPU interrupt request input.
$\overline{\text{RESET}}$	Active-LOW, reset input.

INDEX

A

Accumulator, 13, 21
Accumulator data, 76
Accumulator, MAD, 20
AC-line powered devices, 171
AC-outlets, 125
A/D (see also analog-to-digital converter), 101
A/D converter (see also analog-to-digital converter), 101
A/D conversion, basics of, 145
A/D converter inferfacing, 164
A/D converters, types of, 145
  binary ramp, 145
  dual slope integrator, 145, 153
  flash, 145, 149
  parallel, 145, 149
  servo, 145
  successive approximation, 145, 147
  voltage-to-frequency, 145, 150
ADC (instruction), 211, 215
ADC (see analog-to-digital converter), 101
ADD (instruction), 211–214
Address bus A0-A15, 44
Address bus, MAD, 19
Address bus, 16-bit, 87
Address decoding, 91
Address decoders, I/O port, 62
Address, memory analog (MAD), 19
Addressing modes, 183
Alarm condition, 100
Alarm status, 100
Alphanumeric character, 23
ALU, 13, 43
American Standard Code for Information Interchange (*see also* ASCII), 239
Analog display, 156
Analog-to-Digital Converter, 61, 139
Annunciator panel, 178
Antenna switch box, 25, 29
Apple II, 12, 131
Architecture, Z80, 39
A-register, 13
A-register, MAD, 20
Arithmetic instructions, 193

Arithmetic/logic instructions, 191
Arithmetic/logical operations, 211
Arithmetic Logic Unit, 13
ASCII, 23, 111, 114, 239
ASCII characters, 101
ASCII character input, 114
ASCII code table, 249
ASCII input format, 111
Assembly language, 3
Asynchronous transmission, 77
Atari-style game connector, 127
Audio cassette, 45
Audio signals, 26

B

Bank selection, 92
Bank select terminal, 93
BASIC, 3, 11
BASIC interpreter, 6
BASIC language, 220
BASIC programming, 33
Baud, 79
Baud rate, 77
BAUDOT, 23, 114
BAUDOT code table, 252
BCD-to-1-of-10-decoder, 67
BEEP command, 11
Beta, 175
Bidirectional I/O port circuit, 76
Bidirectional parallel I/O port design, 72
Binary coded decimal, 67
Binary digits, 5
Binary numbers, 22
Binary resistance ladder, 140
Binary weighted resistance ladder, 140
Bit addressing, 184, 190
Bit control, 86
Bit length, 77
Bit manipulation, 191, 194
Block search instruction, 192
Block transfer instruction, 192
Block transfer/search, 191
Bonnie, 35
Breadboard, 131
Buffer, 71
Buffer output, 104
Buffer register transfer completed, 77
Burglar alarm, 100
BUSAK, 49
Bus acknowledge, 49
Bus lines, external, 50
Bus request/acknowledge, 56
Bus request signal, 49
Bus request timing, 56
BUSRQ, 49
Byte array, 91
Byte bidirectional bus, 86
Byte input, 86
Byte output, 85

C

CALL instruction, 194
Carry flag, 59, 215
Carry flag, MAD, 22
Cassette lead, 25
Cassette player, 5
Central processing unit, 13
Central processing unit, MAD, 19
C-flag (see carry flag), 22
Characters, ASCII, 101
Chart recorders, 140
"Chicklet" keyboard, 7
Chip enable, 88
Chip enable pin, 88
Circuits, isolated, 172
Clock pulses, 23
Closed-circuit TV, 28
Code system, BAUDOT, 114
Code system, ASCII, 114
Collector power dissapation, 173
Color computer, 3
Command cartridges, 127, 133
Communications link, 76
Communications port, 80
Comparator, 64
Comparator, voltage, 153
Computer, digital, 13
Computer, single-board, 122
Condition bits, 43
Connectors
  commercial, 131
  DB-25, 119
  DC power, 130

DOCK, 131
Joystick/game, 137
other T/S-2068, 138
RS-232, 119
Constant current source, 151
Constant voltage source, 151
Control logic section, 13
Control logic section, MAD, 21, 22
Control logic signals, MAD, 19
Controlling external devices and circuits, 171
Conversion time, 101, 147
Counterelectromotive force, 172
CPU, 13
CPU, basic timing, 52
CPU chip, 50
CPU control, 191
CPU control instructions, 197
CPU control signals, 51
CPU data bus, 102
CPU instructions sorted by mnemonic, 244
CPU instructions sorted by op-code, 240
CPU, MAD, 19, 21
CPU time, 101
CRT terminals, 118
Current loop, 115
Cycles, MAD, 23

D

DAC (see digital-to-analog converter), 97
DACs, commercial, 143
DACs, multiplying, 157
DAC output circuitry, 163
Darlington amplifier, 176
Darlington pair, 176
Data, 23
Data acquisition, 87
Data/address buses, 49
Data bus, 20
Data bus D0-D7, 44
Data bus, MAD, 19
Data conversion (A/D-D/A), 139
Data latches, 70, 97
Data received terminal, 80
Data storage, 4
DC power connector, 130
DC power supplies, 122
Decoder, BCD, 67
Decoding, address, 91
D-input data, 69
Designer, instrumentation, 157
Device select signals, 66
DIEN line, 73
Digital computer, programmable, 13
Digital displays, 61
Digital-to-analog converter, 61, 139
Dinosaurs, 15
DIP pattern board, 131
Displacement integer, 187
Display, 5
Displays, 177
DOCK connector, 131, 132
Double precision, 161
DRAM, 87
Driver, transistor, 173
Dual-slope integrator, 145
"Dumb" printer, 111
Dual-slope integrator timing, 155
DVM, 156
Dynamic memory, 95
Dynamic Random Access Memory, 87

E

EBCDIC code table, 253
EIA, 118
Electrical lights, 172
Electric typewriters, 114
Electromechanical devices, 101
Electromechanical relays, 172
Electronic Industries Association, 118
Enable line, 104
Environmental control system, 100
End-of-conversion pulse, 146
EOC pulse, 146
EPE, 79
EPROM, 90
Erasable ROM, 87
Exchange instruction, 192
Extended addressing, 183, 188
External circuits, controlling, 171
External devices, controlling, 171

F

FCC, 27
F Register, 43
F′ Register, 43
Filter capacitors, 124
Flag register, 43, 59
Flash A/D, 149
Format bits, 76
Four-bit data latch, 70
Framing error, 77
Full duplex operation, 80
Furnaces, 172
F/V conversion, 151

G

Game connector, Atari style, 127
Generating port/device select signals, 66
Glitch, 121
Glitch witch, 121

H

Half-carry flag, 60
HALT, 48
HALT, exit from, 58
HALT instruction, 48
HALT signal, 48
Handshaking, 113
Hardware methods, 76
Heaters, 172
High voltage circuits, 172
High speed line printers, 101

I

IM1 instruction, 102
Immediate addressing, 183, 184
Immediate extended addressing, 183, 185
Implied register addressing, 184
Implied addressing, 189
Incandescent lamps, 179
Indexed addressing, 183, 188
Index registers IY, IX, 42
Indirect call, 102
Inductive kick, 173
Inductor, 173
Information transfer, 80
Input operation, 66
Input/output, 191
Input/output instruction, 197
Input/output, MAD, 19
Input/output operations, 50
Input/output request, 48
INPUT signal, 68
Instruction decoder, MAD, 22
Instruction fetch, 53
Instructions, I/O, 62
Instruction register, MAD, 22
Instrumentation, electronic, 163
Instruments, factory process, 156
Instruments, medical, 156
INT, 203
INT signal, 48
Integration, 153
Integrator, dual-slope, 145
Interface connector, 14, 135
Interface connector, expanded, 127
Interface connector, pinouts, 129
Interface connector, TIMEX, 127
Interface connector (T/S-1000/1500), 128
Interface connector (T/S-2068), 8
Interfacing, A/D, 164
Interfacing, current loop, 115
Interfacing, DAC, 157
Interfacing, I/O-based, 157
Interfacing, memory, 87
Interfacing, microcomputer, 16, 122
Interfacing, peripheral, 110
Interfacing, RS-232, 118
Interfacing 10-, 12- and 16-bit DACs, 161
Interrupt, 48, 56
Interrupt acknowledge, 104
Interrupt capability, 101
Interrupt flip-flop, 48, 102
Interrupt hardware, 100, 103
Interrupt line, 104
Interrupt, maskable, 101, 203
Interrupt modes, 102
Interrupt, nonmaskable, 48, 101
Interrupt, prioritize, 109

Interrupt request, 56, 101, 103, 104
Interrupt request, maskable, 52
Interrupt response vector, 62
Interrupt, service, 100
Interrupt service subroutine, 101
Interrupts, servicing, 199
Interrupts, types of, 101
Interrupt vector I, 41
Interrupt, what is it?, 100
Inverting line drivers, 72
I/O, 20
I/O based interfacing, 157
I/O based scheme, 166
I/O devices, 61
I/O instructions, 62
I/O, MAD, 19
I/O printed circuit boards, 64
I/O Port, 110, 171
I/O port address decoders, 62
I/O port circuit, 76
I/O read/write, 62
IORQ, 48
I/O write, 62
I-register, 41
Isolated circuits, 172
Isolation transformer, 126

J

Joystick, 137
JUMP, CALL & RETURN, 191
JUMP instruction, 194

K

Keyboard, 6, 61, 101, 239
Keyboard, "chicklet," 7
Keyboard, membrane, 3
Keyboard, T/S-1000, 4
Keyboard, T/S-1500, 7
Keyboard, T/S-2068, 10

L

Lamps, incandescent, 179
Language Oriented Software, 133
Large Scale Integration (LSI), 13
Latch, double buffered, 159
LEDs, 171
LED's, interfacing, 172
LED numerical display, 178
Liquid level detector, 100
Line printer, 101
LOAD and EXCHANGE, 191
Logical instructions, 193
LROS, 133
LSI (see also large scale integration), 13
LSI IC, 15

M

Machine code, entering, 221
Machine cycle-1, 44
Machine language, 3, 221
Machine language programming, 11, 220
Machinery, 172
MAD, 18
MAD computer, 11, 22
Magnetic field, 173
Mainframe computer, 15
Maskable interrupt, 101, 203
Mass data storage, 24
Mass storage, 5
Membrane keyboard, 3
Memory, 13
Memory blocks, 91
Memory chips, 91
Memory chips, solid-state, 95
Memory control signals, 51
Memory data, 87
Memory data read/write, 54
Memory devices, 87
Memory interfacing, 87
Memory location, 97
Memory, MAD, 19
Memory-mapped devices, 97
Memory-mapped I/O, 159
Memory-mapping, 87
Memory operations, 88
Memory read, 87
Memory refresh, 42
Memory request, 48
Memory write, 87

Memory write command, 159
Meter movements, 140
Microcomputer, 13
Microcomputers, advantages of, 15
Microcomputer interfacing, 16
Microcomputer, personal, 12
Microcomputer terminology, 12
Microphone, 6, 24
Microprocessor, 13
Microprocessor fundamentals, 17
Minicomputer, 14
Modes, interrupt, 102
Mode-0, 102, 203
Mode-1, 102, 207
Mode-2, 102, 207
MODEMs, 118
Modified Page-Zero Addressing, 183, 185
Monitor TV, 28
Monitor, video, 7
Monostable multivibrator, 153
Motors, 140
MOV device, 125
MREQ, 48
Multiplex, data, 161
Mythical Analytical Device, 11
M1 period, 50
M1 signal, 44

N

NMI signal, 48
NMI, 199
Noninterrupt operation, 106
Noninverting line drivers, 72
Nonmaskable interrupt, 48, 57, 101, 199, 202
Numerical display, 178
Numerical methods, 23
Numerical readouts, 171
Nyquist, 165

O

Ohm's law, 141
On-off switch, 9
Op-codes, 23
Op-code fetch, 53
Operational amplifier, 140
Operation codes, 23
Operations, I/O, 50
Optoisolator, 117, 172, 177
Optotransistor, 117
Oscilloscope, 156
Output command, 159
Output operation, 67
Overrun error, 77
Over temperature alarm, 100
Overvoltage protection, 125

P

Package configuration, Z80, 45
Panel, annunciator, 178
Parallel A/D, 149
Parallel input port design, 71
Parallel In Serial Out, 76
Parallel I/O ports, 76
Parallel output port design, 69
Parity, 77
Parity error, 77
Parity inhibit, 77
PC, 41
PC contents, 57
PC, MAD, 21
PCD (see precision charge dispenser), 153
PEEK, 221
Peripheral, 101, 239
Peripheral device signal, 101
Peripherals, interfacing, 110
PISO, 76
Plug-in I/O cards, 69
POKE, 221
Port Select signals, generating, 66
Power supply, 24, 117
Power supply, DC regulated, 122
Power supply filter, 124
Power supply module, TIMEX, 121
Precision Charge Dispenser (PCD), 153
Precision reference voltage source, 153
Printers, 61, 113, 118, 239
Printed cards, prototyping, 131
Printer driver routines, 76
Printer, "dumb," 111

Program counter, 41, 42
Program counter, MAD, 21
Program, writing a, 34
Programmable digital computer, 13
Programmable Read Only Memory, 87
Programmer, 102
Programming, 34
Programming pins (UART), 80

R

Radio link, 76
RAM, 6, 11
RAM dynamic, 87
RAM, internal, 8
RAM, static, 87
Random access memory, 6
RC oscillators, 79
RC timer-based clocks, 79
RD signal, 48
Read Only Memory, 16, 87
Read signal, 48
Real time clock, 100
Received data available, 77
Receiver circuit, 80
Receiver section, 77
Receiver verification, 77
Recorder, audio, 4
Refresh signal, 48
Registers, 13
Register addressing, 184, 189
Registers, general purpose, 41
Register indirect addressing, 184, 189
Register IX, 41
Register IY, 41
Regulator, DC voltage, 125
Relative addressing, 183
Relative addressing, 187
Relay, 173
REM statement, 221
RESET, 48
Resistance ladder, binary, 140
Resistance ladder, R-2R, 142
Resistor network, 149
Restart address, 102
Restart instruction, 102
RETURN instruction, 194
RF modulator, 5
RFSH, 48
ROM, 11
ROM, bank select, 11
ROM cartridge, 128
ROTATE instruction, 194
ROTATE and SHIFT, 191
R-register, 42
RST op-code, 106
RS-232 B-vs-C, 119
RS-232 connector, 119
RS-232 interfacing, 118
RS-232 pin definitions, 120
R-2R resistance ladder, 142

S

SBC instruction, 211
SBS, 79
Schmitt trigger, 151
SELECT signal, 64
Semiconductors, 172
Semiconductor power devices, 125
Serial communications, 76
Serial In Parallel Output, 77
Serial input word, 79
Serial I/O port, 76
Servicing interrupts, 199
Seven bar type LED's, 179
Seven-bit code, 111, 239
Seven-segment LED display, 179
SHIFT instruction, 194
Sign flag, 59
Simplex communications channel, 80
Sinclair ZX-80, 3
Single-board computer, 122
Single-chip computer, 13
SIPO, 77
60-mA current loop, 115
Smoke detector, 100
Software cartridges, 10
Software serial I/O port, 76
Solenoids, 114
Solenoid coils, 116
SP, 41
Spike suppressor, 116
Stack pointer, 41, 42
Start bit, 76
Start pulse, 148

Status flags, MAD, 22
Status register, MAD, 21, 22
Stop bit, 76
Stop bit length select, 77
Strip-chart recorder, 156
Strobe, 239
Strobe terminals, 97
SUB instruction, 211, 215–219
Subroutine, interrupt service, 101
Support chips, Z80, 255
Synchronous transmission, 77
System I/O signals, 69

T

Tape recorder, 4, 5
Telephone, 14
Telephone link, 76
Telephone systems, 80
Teletypewriters, 61, 76, 101, 114
Television, 5
Television displays, ordinary, 10
Television receiver, 26
Terminal, 101
Timer, 100
Timing signal, MAD, 20
TIMEX interface connector, 127
TIMEX T/S-1000, 3
TIMEX T/S-1500, 7
Touchtone[R], 14
Transducer, 156
Tranducible events, 156
Transformer, 122
Transformer, isolation, 126
Transmission, asynchronous, 77
Transmission completed, 77
Transmission, synchronous, 77
Transmitter circuit, 80
Transmitter generation, 77
Transmitter section, 77
Trigger threshold, Schmitt, 153
Tri-state buffer, 104
Tri-state device, 71
Tri-state mode, 104
Tri-state output lines, 71
TRS-80, 12
T/S-1000, connecting the, 30
T/S-2068, 8
TTL-compatible, 110
TV antenna switchbox, 25
TV, closed circuit, 28
TV, monitor, 28
TV output, 24
TV screen, 171
20-mA current loop, 115
Type-D flip-flop, 69
Typewriters, 114

U

UART, 76, 171
UART block diagram, 78
UART data format, 79
Universal Asynchronous Receiver/Transmitter, 76 (*see also* UART)
μC, 13
μP, 13
USR subroutine, 220

V

VCO, 150
Vectored interrupt, 102
Vector, interrupt response, 62
Video, 5
Video monitor, 7
Voltage controlled oscillator (VCO), 150, 151
Voltage comparator, 153
Voltage regulator, 125
Voltage-to-Frequency Converter (VFC), 150

W

WAIT, 48
WAIT signal, 48
WAIT states, adding, 95
WLSI and 2, 79
Word Length Select, 79

WR, 48
Write signal, 48

Z

Zero flag, 59
Zilog, 7
ZX-81, 6
Z80 addressing modes, 183
Z80 architecture, 39
Z80 computer, 49
Z80 control signals, 49
Z80-CTC, 255
Z80 family support chips, 255
Z80 flags, 58
Z80 instructions sorted by mnemonic, 244
Z80 instructions sorted by op-code, 240
Z80 interrupt, types of, 101
Z80 I/O control signal, 61
Z80 I/O instructions, 62
Z80 instructions, 183
Z80 microprocessor chip, 6
Z80 package configuration, 45
Z80 pin definitions, 44
Z80-PIO, 84, 255
Z80-SIO, 255
Z80 timing signals, 45
Z80 users manual, 20

BookWare *For the Timex Sinclair 2068, 1500 and 1000 Computers*

Timex Sinclair 2068, 1500 and 1000 Machine Language Programming and Interfacing
Joseph J. Carr

An excellent, all-in-one guide to machine language programming on Timex Sinclair computers. You'll get step-by-step instructions on how to program the popular Z80 chip. Joseph Carr provides you with a number of topics, all covered in-depth, to get you up and running fast on the surprisingly powerful Timex Sinclair machines. You'll find the book packed with information on: Z80 architecture; interfacing memory to the Z80; interfacing peripherals; data conversion; servicing interrupts; and much more. By using machine language you'll find that you're writing much more efficient programs, and best of all, programs that take much less memory than programs written in BASIC do. In fact, Timex Sinclair computers become even more powerful by using machine code! So read through Joseph Carr's book and find out more about **Timex Sinclair 2068, 1500 and 1000 Machine Language Programming and Interfacing.**

Also From Reston . . .

Mastering Machine Code on Your Timex Sinclair 1500/1000
Toni Baker

The first simple guide to machine code on the popular Timex Sinclair 1500 computer. Filled with excellent programs, illustrations, and plenty of examples, this comprehensive guidebook gives you all the programming techniques you'll need. You'll find many surprising and little-known tricks—like a scroll program to make the screen scroll downwards instead of upwards. You'll also find graphics, games, debugging techniques, arithmetic subroutines, and much more. If you want to learn how to get the most out of your computer, this is the book for you!

BASIC Basics for the Timex Sinclair 1500/1000
Michael P. Barnett/Simon J. Barnett

This is an excellent way to introduce your family to BASIC on your Timex Sinclair 1500 and ZX-81 computer. You'll find it filled with useful information—for managing household finances, using your Timex Sinclair 1500 as an electronic message board, using the computer as a calculator, solving complicated mathematical problems, and many other applications.

A Reston Computer Group Book
RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

0-8359-7699-8